

Todos os novos funcionários da *Facedata* são obrigados a passar pela adaptação e a parte mais interessante é um curso intensivo de Python. Este não será um tutorial abrangente mas que pretende destacar as partes da linguagem que serão mais importantes para desenvolver nosso trabalho.

Considerações Iniciais

Algumas premissas importantes sobre estética em linguagens de programação em geral são:

- Bonito é melhor que feio.
- Explícito é melhor que implícito.
- Simples é melhor que complexo.
- Complexo é melhor que complicado.
- Legibilidade faz diferença.

E representam os ideais pelos quais faremos nossos códigos (ou *scripts*).

Embora, nós, humanos programamos códigos para as máquinas entenderem, é de extrema importância que outros humanos também os compreendam. Esse passo é tão importante que a gigante da tecnologia *Google* fornece um [guia de estilo de programação \(em inglês\)](#) para diversas linguagens. Vamos seguir várias das convenções lá sugeridas.

A conclusão do guia de estilo da linguagem Python é bem direta e diz:

SEJA CONSISTENTE.

DICAS VALIOSAS PARA QUALQUER LINGUAGEM DE PROGRAMAÇÃO

- **Use (e abuse) de documentação:** Toda linguagem de programação possui formas para inserir um texto que não é interpretado pela máquina mas é essencial para comentar o que estamos fazendo ali;
- **Não use acentos:** Este é campo nebuloso e não vamos entrar muito em detalhes. Cada computador, dependendo da forma em que está configurado, pode entender a palavra `benção` como `bÃªnÃ§Ã£o` ou como `b?n?o`, por exemplo. Opte por `bencao`;
- **Letras maiúsculas e minúsculas:** Praticamente todas as linguagens de programação distinguem letras maiúsculas e minúsculas. Por exemplo, `Inteligencia Artificial` \neq `inteligencia artificial`. Tenha preferência para o uso geral de letras minúsculas. Maiúsculas somente quando muito necessário;
- **Não use espaço para definição de objetos:** Espaços são extremamente importantes para a clareza (principalmente pelos humanos) de linhas de comando. No entanto, um objeto qualquer que se chama `inteligencia artificial` será compreendido como 2 objetos (`inteligencia` e `artificial`), ocasionando erros. Opte por `inteligencia_artificial`;

- **Inglês:** As mais importantes linguagens são documentadas em inglês. Dominar a leitura de textos em língua inglesa só facilitará seu desenvolvimento. Além disso, temos muitas palavras reservadas dentro de cada linguagem para simbolizar algum comando, por exemplo `True` (Verdadeiro) e `False` (Falso).

A Linguagem Python

Python é uma linguagem de programação para propósitos diversos e pode ser usada para um enorme leque de aplicações em diferentes áreas, desde o desenvolvimento *web* até o aprendizado de máquina. Além de versátil, esta linguagem é amigável para iniciantes. Por esses e outros fatores, trata-se de uma das linguagens de programação mais populares do mundo.

Embora Python (Píton em português) seja o gênero de serpentes da família *Pythonidae*, este nome foi dado em homenagem ao humorístico britânico *Monty Python* que fez sucesso na década de 1970.



Durante o aprendizado de programação, iniciantes podem encontrar alguma dificuldade ao tentar entender como uma determinada linguagem funciona, especialmente se ela é muito diferente do seu idioma nativo. Mas fique tranquilo(a)! Honrando sua distinção como uma das linguagens mais fáceis de se aprender, o Python tem sintaxe muito simples, com muitas palavras-chave em inglês. Ele foi projetado desde o início para ser uma linguagem concisa,

de alta legibilidade e fácil compreensão. Por isso, o Python é muito mais amigável para iniciantes quando comparado à outras linguagens.

Tutorial

Este (mini) tutorial o guiará pela compreensão da linguagem de programação Python, ajudará você a aprender profundamente os conceitos e mostrará como aplicar (algumas) técnicas práticas de programação aos nossos desafios na *Facedata*.

Vamos lá!

Calculadora

Você pode utilizar o Python como uma grande calculadora:

```
In [ ]: 6 + 3
```

```
In [ ]: 6 - 3
```

```
In [ ]: 6 * 3
```

```
In [ ]: 6 / 3
```

Use (somente) parênteses para a realização de contas complexas como

$$3 \times \{12 + [(3 + 2) - 5 \times 3] \times 5\} :$$

```
In [ ]: 3 * (12 + ((3 + 2) - 5 * 3) * 5)
```

Note que no trecho do cálculo

$$\begin{aligned} [(3 + 2) - 5 \times 3] &= [5 - 5 \times 3] \\ &= [5 - 15] \\ &= -10, \end{aligned}$$

O Python segue a ordem matemática das operações, priorizando a multiplicação (ou divisão) e depois, a subtração (ou adição).

Os operadores aritméticos básicos disponíveis no Python são:

| Operador | Conceito | Exemplo |
|----------|-----------------------------|----------|
| + | Adição ou sinal positivo | 13 + 7 |
| - | Subtração ou sinal negativo | - 13 - 7 |
| * | Multiplicação | 3 * 4 |
| / | Divisão | 10 / 5 |
| // | Divisão inteira | 10 // 6 |

| Operador | Conceito | Exemplo |
|----------|---------------|---------|
| % | Módulo | 4 % 2 |
| ** | Exponenciação | 4 ** 2 |

Objetos

Podemos salvar as contas que fazemos em objetos que serão chamados de *variáveis*:

```
In [ ]: a = 6 + 3
```

```
In [ ]: b = 6 - 3
```

```
In [ ]: c = 6 * 3
```

```
In [ ]: d = 6 / 3
```

Para visualizá-los, basta utilizar o comando `print` :

```
In [ ]: print(a)
```

```
In [ ]: print(b)
```

Podemos ainda realizar operações quando os objetos (ou as variáveis) são do mesmo tipo. Ou seja, aqui ainda estamos tratando somente de números. Então, podemos fazer contas diretamente:

```
In [ ]: print(a + b)
print(3 * b)
print(3 * a + 2 * b + 5 * c + 4 * d)
```

Tipos

Numéricos

Quando tratamos de números no Python, internamente a máquina pode entendê-los de duas formas:

- `float` : Formato numérico com casas decimais. Ex.: `1` é entendido pelo computador como `1.00000000...` ;
- `int` : Formato numérico sem casas decimais (inteiros). Ex.: `3.145` é entendido pelo computador como `3` .

O Python utiliza o padrão numérico estadunidense onde o símbolo para separar a parte inteira da decimal de um número é o **ponto** · (diferente do padrão brasileiro que utiliza **vírgula**).

```
In [ ]: a = float(22/5)
b = int(4.5)
```

```
c = int(3.9)

print(a)
print(b)
print(c)
```

Cuidado ao somar esses 2 tipos:

```
In [ ]: print(a + b)
```

```
In [ ]: type(a + b)
```

Alfanuméricos

O Python é muito poderoso e consegue lidar com outros tipos de dados além dos números. Temos disponíveis operadores *booleanos*. Os valores do tipo `bool` podem representar dois valores completamente distintos: `True` (igual ao `int 1`) e `False` (igual ao `int 0`) para, respectivamente, `verdadeiro` e `falso`:

```
In [ ]: operador_bool_verdadeiro = True
operador_bool_falso = False
```

Note que a primeira letra é maiúscula! Ortografias distintas não são permitidas.

Além disso, podemos utilizar os operadores lógicos que resultam em resultados do tipo `bool`:

| Operador | Conceito | Exemplo |
|--------------------|----------------|----------------------------|
| <code>==</code> | Igual | <code>5 == (10 / 2)</code> |
| <code>!=</code> | Diferente | <code>1 != 0</code> |
| <code>></code> | Maior | <code>3 > 1</code> |
| <code>>=</code> | Maior ou igual | <code>6 >= 4</code> |
| <code><</code> | Menor | <code>5 < 2</code> |
| <code><=</code> | Menor ou igual | <code>2 <= 3</code> |

```
In [ ]: a = 3
b = 5

print(a > b)
print(a == (b - 2))
print((a * 5) != (b * 3))
```

Não confunda:

- `=` (símbolo de **atribuição**);
- `==` (símbolo de **igualdade**).

Por fim, temos o tipo `string` e é com ele que tratamos texto delimitando por aspas simples ou duplas (mas as aspas devem corresponder):

```
In [ ]: string_aspas_simples = 'ciencia de dados'
        string_aspas_duplas = "ciencia de dados"

        print(string_aspas_simples)
        print(string_aspas_duplas)
```

Tome cuidado ao somar `strings` e `floats`. Veja o que acontece:

```
In [ ]: nome = 'Dino'
        sobrenome = "da Silva Sauro"
        idade = 43

        print(nome + sobrenome + idade)
```

Quando os tipos não são os mesmo, devemos usar o seguinte comando para correta impressão:

```
In [ ]: print(f"Nome completo: {nome} {sobrenome}. Idade: {idade}")
```

Documentação

Fazemos a documentação diretamente no código utilizando comentários. Assim, nosso código fica mais legível para outros humanos. Além disso, os comentários também são úteis para fazer com que o computador não leia determinado bloco durante um teste, por exemplo. Em Python, os comentários são simbolizados pelo símbolo `#` e se estende até o fim de cada linha. Vamos documentar os blocos anteriores:

```
In [ ]: # Nome do cliente
        nome = 'Dino'

        # Sobrenome do cliente
        sobrenome = "da Silva Sauro"

        # Idade do cliente
        idade = 43          # valor numerico

        # Imprimindo um texto na tela
        print(f"Nome completo: {nome} {sobrenome}. Idade: {idade}")
```

Funções

Uma função é uma regra que recebe entradas (que são chamadas de argumentos) e retorna uma saída correspondente. Embora não tenhamos mencionado, mas já utilizamos duas funções, são elas `print` e `type`. Mas podemos definir novas funções em Python usando `def`:

```
In [ ]: # Funcao que calcula o dobro de um numero
        def dobro(x):
            """Insira aqui um texto documental (opcional) que
            explique o que a funcao faz. Por exemplo, esta funcao multiplica sua
            entrada por 2.
            """
            return x * 2
```

```
In [ ]: # Testando a nova funcao
print(dobro(5))
print(dobro(-5))
print(dobro(.5))
```

A linguagem Python usa **indentação** (espaçamento) para delimitar os blocos de código. Note que todo o conteúdo da função `dobro` possui um espaçamento. Isso torna o código legível e organizado.

Podemos elaborar e aprimorar as saídas das funções:

```
In [ ]: # Funcao que calcula o salario anual
def salario_anual(salario_mensal):
    """Funcao que calcula o salario anual
    (considerando o 13o salario)
    """
    return print(f"O salario anual e de R$ {salario_mensal * 13}")
```

```
In [ ]: # Teste
salario_anual(2500)
```

Os argumentos de uma função podem receber padrões que só precisam ser especificados quando você deseja um valor diferente do padrão estabelecido:

```
In [ ]: # Funcao que escreve o nome completo do cliente
def nome_completo(nome = "Nome", sobrenome = "Sobrenome"):
    return nome + " " + sobrenome
```

```
In [ ]: # Testes
print(nome_completo('Dino', 'da Silva Sauro')) # 'Dino da Silva Sauro'
```

```
In [ ]: print(nome_completo(sobrenome = 'da Silva Sauro',
                             nome = 'Dino')) # 'Dino da Silva Sauro'
```

```
In [ ]: print(nome_completo('Dino')) # "Dino Sobrenome"
```

```
In [ ]: print(nome_completo(sobrenome = 'da Silva Sauro')) # "Nome da Silva Sauro"
```

Nós criaremos muitas funções ao longo do curso (e da vida profissional)!

Estrutura condicional

Você pode executar ações de forma condicional utilizando `if` e `else`

Em português, `if` significa "se" e `else` significa "senão":

```
In [ ]: ##### Exemplo (Nao rode) !!!!!
if 1 > 2:
    # Se 1 e maior que 2, faca...
elif 1 > 3:
    # 'elif' significa 'else if' (senao se...) [opcional]
    # Se 1 e maior que 3, faca ...
else:
    # Caso todas as condicoes sejam falsas, faca... (opcional)
```

```
In [ ]: # Funcao que calcula o salario anual
def salario_anual(salario_mensal):
    """Funcao que calcula o salario anual
    (considerando o 13o salario)
    """
    if salario_mensal > 0:
        resultado = f"O salario anual e de R$ {salario_mensal * 13}"
    else:
        resultado = "O salario deve ser um valor positivo"

    return print(resultado)
```

```
In [ ]: # Teste
salario_anual(-2500)
salario_anual(2500)
```

Podemos também criar *loops* (ou laços) utilizando `for` e `in` para \textit{iterar} valores:

Em português, `for` significa "para" e `in` significa "em".

```
In [ ]: # range(k) e uma funcao que cria uma sequencia de 0 ate (k - 1).
# Exemplo: range(10) sera a sequencia 0, 1, 2, ..., 9

for x in range(10):
    print(x + " e menor do que 10")
```

Voltaremos a falar do uso de `for` e `in` em contextos mais interessantes.

Listas

Provavelmente, a estrutura de dados mais fundamental do Python é a **lista**, que é simplesmente uma coleção ordenada (é semelhante ao que em outras linguagens pode ser chamado de `array`, mas com algumas funcionalidades adicionais):

```
In [ ]: lista_numeros = [1, 2, 3]    # somente numeros (floats)
```

```
In [ ]: lista_mista = ["string", 0.1, True] # string, float, bool
```

```
In [ ]: # combinando as 2 listas anteriores com uma vazia
lista_de_listas = [lista_numeros, lista_mista, []]
```

```
In [ ]: comprimento_lista = len(lista_numeros) # 3 elementos
```

```
In [ ]: soma_lista = sum(lista_numeros) # 6 (soma dos elementos numericos)
```

Você pode obter ou definir o n -ésimo elemento de uma lista com colchetes:


```
In [ ]: x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [ ]: zero = x[0] # primeiro elemento de x - listas comecam com indice zero!
```

```
In [ ]: um = x[1] # segundo elemento de x
```

```
In [ ]: nove = x[-1] # ultimo elemento de x
```

```
In [ ]: oito = x[-2] # penultimo elemento de x
```

```
In [ ]: x[0] = -1 # agora x e [-1, 1, 2, 3, ..., 9]
```

A linguagem Python inicia seu contador em **ZERO**!

Por exemplo, o **primeiro elemento** de uma lista tem **índice 0**. De forma semelhante, o **décimo elemento** tem **índice 9**.

Você também pode usar colchetes para *fatiar* listas. A fatia `i:j` significa todos os elementos de `i` (inclusivo) a `j` (exclusivo).

```
In [ ]: primeiros_tres = x[:3] # [-1, 1, 2]
```

```
In [ ]: quarto_em_diante = x[3:] # [3, 4, ..., 9]
```

```
In [ ]: segundo_ate_quinto = x[1:5] # [1, 2, 3, 4]
```

```
In [ ]: ultimos_tres = x[-3:] # [7, 8, 9]
```

```
In [ ]: sem_primeiro_e_ultimo = x[1:-1] # [1, 2, ..., 8]
```

```
In [ ]: copia_de_x = x[:] # [-1, 1, 2, ..., 9]
```

Uma *fatia* pode receber um terceiro argumento para indicar seu passo, que pode ser negativo:

```
In [ ]: a_cada_tres = x[::3] # [-1, 3, 6, 9] -> primeiro, quarto, setimo...
```

```
In [ ]: sexto_ao_quarto = x[5:2:-1] # [5, 4, 3]
```

Dispomos também do operador `in` para verificar se determinado elemento pertence à uma lista:

```
In [ ]: print(1 in [1, 2, 3]) # True
```

```
In [ ]: print(0 in [1, 2, 3]) # False
```

Retornando a parte de nossa base de dados de clientes, podemos *iterar* sobre a lista de nome dos clientes:

```
In [ ]: # Lista de clientes
clientes = ["Nero", "Atum", "Bois", "Alvares"]
```

```
In [ ]: # Iterando ID e nome dos clientes
for id, nome in enumerate(clientes):
    print(f"O id {id} pertence ao cliente {nome}")
```

Programação Orientada à Objetos

A linguagem Python permite que você defina classes que encapsulam dados e funções que operam dentro delas. Tais funções são conhecidas como **métodos**. Usualmente, não iremos criar novas classes durante o curso (mas você pode). Iremos estudar os métodos das principais classes (ou como chamamos previamente, "tipos").

Os métodos são acessados através de um ponto · após o nome do objeto. As opções disponíveis de métodos depende da classe (ou tipo) do objeto. Vamos explorar alguns métodos relacionados às `strings`:

```
In [ ]: # String
info_cliente = "Dino da Silva Sauro gosta de ler livros de ficcao cientifica"
```

```
In [ ]: # Metodo find: retorna em qual caracter se inicia a string 'Sauro'
print(info_cliente.find('Sauro')) # 14
print(info_cliente.find('Texto_inexistente')) # -1 (nao contem)
```

```
In [ ]: # Metodo replace: substitui uma string por outra
info_cliente.replace('Dino', 'Nome')

# Note que o resultado nao e salvo
print(info_cliente)
```

```
In [ ]: # Metodo split: cria uma lista separando cada string
info_cliente.split()
```

```
In [ ]: # Metodo upper: todas as letras maiusculas
info_cliente.upper()
```

```
In [ ]: # Metodo lower: todas as letras minusculas
info_cliente.lower()
```

```
In [ ]: # Metodo strip: retira espacos em branco extra do inicio e final
txt = "        muito espaco branco antes e depois        "
txt.strip()
```

Esses são somente alguns métodos para `strings`. Existem vários outros.

Em programação, constantemente você deverá consultar ajuda ou manuais. Mais importante do que conhecer todas as funções (você não conseguirá... sempre há novidade neste meio) é saber procurar ou a lógica do que se deseja fazer.

Vamos aprender também sobre alguns métodos relacionados à listas:

```
In [ ]: # Lista de clientes
clientes = ["Nero", "Atum", "Bois", "Alvares"]
```

```
In [ ]: # Metodo append: adiciona elementos ao fim da lista (um de cada vez)
        clientes.append("Teatro")
        clientes.append("Zinco")
        clientes.append("Ameis")
```

```
In [ ]: # Note que este metodo ALTERA a lista original
        print(clientes)
```

```
In [ ]: # Metodo extend: adiciona uma lista ao fim da lista
        clientes.extend(["Bete", "Biscoito", "Love"])
```

```
In [ ]: # Note que este metodo ALTERA a lista original
        print(clientes)
```

```
In [ ]: # Metodo insert: adiciona um elemento em uma posicao especifica
        # Neste caso na TERCEIRA posicao (indice 2)
        clientes.insert(2, "Intruso1")
```

```
In [ ]: # Neste caso na OITAVA posicao (indice 7)
        clientes.insert(7, "Intruso2")
```

```
In [ ]: # Note que este metodo ALTERA a lista original
        print(clientes)
```

```
In [ ]: # Metodo index: retorna o indice de um elemento pre determinado
        print(clientes.index("Intruso1"))
        print(clientes.index("Intruso2"))
```

```
In [ ]: # Metodo pop: remove um elemento em uma posicao especifica
        clientes.pop(2)

        # Note que este metodo ALTERA a lista original
        print(clientes)
```

```
In [ ]: # Metodo remove: remove um elemento pre determinado
        clientes.remove("Intruso2")

        # Note que este metodo ALTERA a lista original
        print(clientes)
```

```
In [ ]: # Metodo reverse: inverte a ordem da lista
        clientes.reverse()

        # Note que este metodo ALTERA a lista original
        print(clientes)
```

```
In [ ]: # Metodo sort: ordena a lista
        clientes.sort()

        # Note que este metodo ALTERA a lista original
        print(clientes) # Para strings, a ordem e alfabetica
```

```
In [ ]: numeros = [2, 5, 1, 3, 4, 8, 10]
        print(numeros)
```

```
In [ ]: numeros.sort()
        print(numeros) # Para numeros, a ordem e crescente
```

```
In [ ]: numeros.sort(reverse = True)
        print(numeros) # Ou decrescente
```

```
In [ ]: # Lista de interesses
        interesses = ["Python", "Matematica", "IA", "IA", "Python", "Dados", "IA"]

        # Metodo count: retorna a contagem de determinado elemento
        print(interesses.count("Python"))
        print(interesses.count("IA"))
        print(interesses.count("Dados"))
```

```
In [ ]: # Metodo clear: Limpa a Lista
        interesses.clear()

        # Note que este metodo ALTERA a Lista original
        print(interesses)
```

Por fim e não menos importante, usualmente, não modificamos a lista original. Em geral, fazemos cópias do mesmo objeto para trabalhar separadamente com cada um deles. Pode ser tentador programar `lista1 = lista2`. No entanto, isso **NÃO** deve ser feito pois o Python continua referenciando a `lista1`. Ou seja, tudo que você faz em `lista2` é espalhado em `lista1`. Como solução, basta utilizar o seguinte método:

```
In [ ]: # Metodo copy: retorna uma copia da lista
        clientes_novo = clientes.copy()

        # Adicionando um registro na nova lista
        clientes_novo.insert(0, "Intruso3")

        # Comparando
        print(clientes)
        print(clientes_novo)
```

Módulos

Certas funcionalidades do Python não são carregadas por padrão. Isso inclui funcionalidades que já vem inclusas como parte da linguagem, bem como funcionalidades de terceiros que você mesmo faz o *download*. Para usar esses recursos, você precisará importar os *módulos* que os contêm. Uma abordagem é simplesmente importar o próprio módulo:

```
In [ ]: # Importando modulo
        # (ou carregando biblioteca)
        # (ou carregando pacote)
        import statistics

        print(statistics.mean([1, 2, 3, 4, 5]))
```

Aqui, `statistics` é o módulo contendo funções e métodos para cálculos estatísticos. Após esse tipo de importação, você deve prefixar essas funções com `statistics.` para acessá-las.

Por exemplo, se você já tem um `statistics` em seu código, você pode 'apelidar' o módulo da seguinte forma:

```
In [ ]: import statistics as stats
print(stats.mean([1, 2, 3, 4, 5]))
```

Você também pode fazer isso se o seu módulo tiver um nome complicado ou se for digitá-lo muitas vezes. Por exemplo, uma convenção padrão para manipulação de dados com `pandas` e visualização de dados com `matplotlib` é:

```
In [ ]: import pandas as pd
import matplotlib.pyplot as plt

# Exemplos de uso
# pd.read_excel(...)
# plt.plot(...)
```

Se você precisa de funcionalidades específicas de um módulo, poderá importá-las explicitamente e usá-las sem prefixação:

```
In [ ]: from math import ceil

# Exemplos de uso
# (sem necessidade de escrever 'math' antes da funcao 'ceil')
ceil(5.8)
```

Cuidado ao importar todo o conteúdo de um módulo para seu ambiente de trabalho, o que pode substituir inadvertidamente variáveis ou funções que você já definiu:

```
In [ ]: prod = 10

from math import *    # opa, 'math' tem uma funcao que se chama 'prod'
print(prod)           # "<built-in function prod>"
```

Bem-vindo(a) à *Facedata*!

Concluimos nosso treinamento inicial. A partir de agora, você já tem o conhecimento das ferramentas. Agora, vamos colocar a mão na massa!

Referências Adicionais

- Há muitos tutoriais de Python disponíveis na internet, inclusive em vídeo. Embora em inglês, este [tutorial da W3 School](#) é dinâmico e de simples aprendizado;
- O tutorial do [Google Colab](#) também é bem útil e está disponível em português.