Aula 05 - Sintaxe da Linguagem Python

O Zen do Python

Como dito em aulas anteriores, o Python foi construído para ser uma linguagem fácil e rápida de ser escrita.

Como ocorre com vários produtos, sobretudo da área da tecnologia, o Python tem o que na Indústria é chamado de ***Easter Egg*** (Ovo de Páscoa, em inglês), que nada mais é do que um segredo escondido que apenas alguns conseguem encontrar. Esse *Easter Egg*, ao ser encontrado, revela toda a filosofia por trás do funcionamento do Python, que é chamado de **Zen do Python**. Para encontrá-lo, basta executar o código `import this`, que se encontra logo abaixo:

In [ ]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

A saída desse código é nada mais do que um poema com os mandamentos do Python. Segue abaixo a tradução desse poema:

O Zen do Python, de Tim Peters

Bonito é melhor que feio.
Explícito é melhor que implícito.
Simples é melhor que complexo.
Complexo é melhor que complicado.
Plano é melhor que aninhado.

Esparso é melhor que denso.
A legibilidade conta.
Casos especiais não são especiais o suficiente para quebrar as regras.
Embora a praticidade supere a pureza.
Os erros nunca devem passar silenciosamente.
A menos que seja explicitamente silenciado.
Diante da ambiguidade, recuse a tentação de adivinhar.
Deveria haver uma - e de preferência apenas uma - maneira óbvia de fazer isso.
Embora essa forma possa não ser óbvia à primeira vista, a menos que você seja holandês.
Agora é melhor do que nunca.
*Embora nunca seja melhor do que *agora*.*
Se a implementação for difícil de explicar, é uma má ideia.
Se a implementação for fácil de explicar, pode ser uma boa ideia.
Namespaces são uma ótima ideia - vamos fazer mais deles!

Saída de dados do Python

Na aula anterior, vimos o comando `print('Olá, mundo!')`, que é utilizado para mostrar uma saída de dados da linguagem no console, que nesse caso foi a mensagem **Olá, mundo!**.

Nas versões anteriores do Python, esse comando podia ser executado sem os parênteses, simplesmente executando `print 'Olá, mundo!'`, mas isso deixou de funcionar a partir do Python 3.x. O texto a ser exibido pode ser colocado entre aspas simples (') ou duplas ("). Ou seja, o comando pode ser executado tanto com `print('Olá, mundo!')` como com `print("Olá, mundo!")`. Em ambos os casos, irá funcionar perfeitamente. Repare que os comandos em Python nunca são finalizados com ponto e vírgula (;).

O código com aspas simples:

In [ ]:
```
print('Olá, mundo!')
```

```
Olá, mundo!
```

O mesmo código com aspas duplas:

In [ ]:
```
print("Olá, mundo!")
```

```
Olá, mundo!
```

Comentários

Os comentários são pedaços de código que não são executados. Sua função é basicamente conter alguma informação sobre uma parte do código-fonte, auxiliando o programador a se lembrar daquela funcionalidade, ou ajudando um possível substituto a entender seu código e ajudá-lo a continuar o algoritmo. Em Python, o comentário de

uma linha é feito usando o símbolo de serquilha/hashtag/jogo da velha/sustenido/sharp (#). Já os comentários de múltiplas linhas é iniciado com três aspas simples ou duplas (''') ou ("""), e finalizado também com 3 aspas (simples se tiver iniciado com simpels, ou duplas se tiver iniciado com duplas).

Exemplos:

In [ ]:
```
# Este é um exemplo de um comentário de uma linha
```

In [ ]:
```
'''
Este é um exemplo
de um comentário
de múltiplas linhas
com aspas simples
'''
```

In [ ]:
```
"""
Este é um exemplo
de um comentário
de múltiplas linhas
com aspas duplas
"""
```

Anchor/TODO Comments

Alguns editores/IDEs podem identificar comentários específicos, que funcionam como âncoras ou marcadores no seu código-fonte. Esses comentários têm como objetivo indicar alguma observação mais específica em determinado trecho de código, como uma funcionalidade a ser acrescentada ou corrigida, por exemplo. São sempre escritos em caixa alta. Segue os exemplos mais usados abaixo:

TODO

Serve para marcar um trecho a fazer:

In [ ]:
```
# TODO: fazer esse trecho de código
```

FIXME

Serve para marcar um trecho de código defeituoso, que está causando um *bug* ou impede a execução do código, e que deve ser corrigido:

In [ ]:
```
# FIXME: corrija esse trecho de código
printar(texto para ser exibido);
```

```
  Cell In[9], line 2
    printar(texto para ser exibido);
            ^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

REVIEW

Serve para marcar um trecho para ser revisado. Normalmente o código aparentemente funciona, mas pode ser que esteja causando um *bug* em outra parte do código, ou pode ser um algoritmo que pode ser melhorado:

In [ ]:
```
# REVIEW: revise esse código
texto1 = 'Exibir'
texto2 = 'este texto'

print(texto1)
print(texto2)
```

```
Exibir
este texto
```

NOTE

Serve para informar algum aviso importante sobre um trecho do código:

In [ ]:
```
# NOTE: você pode executar esse código clicando no botão Executar do lado dessa
print('Código rodado!')
```

```
Código rodado!
```

Variáveis

Em Álgebra, aprendemos o conceito de **variável**, que nada mais é do que um valor, que como o próprio nome já diz, varia, e não podemos ter certeza de qual valor ele vai assumir. Geralmente são os valores dessas variáveis que queremos descobrir em uma equação matemática.

Na programação usamos esse mesmo conceito para guardar um valor que iremos utilizar em algum momento dentro do nosso algorítmo, sendo que não sabemos com certeza que valor é esse, e que pode ser modificado no decorrer do nosso programa a qualquer momento.

Mas como esse valor é guardado?

Imagine a seguinte situação: você quer fazer um jantar super romântico para a sua namorada para comemorar o aniversário de namoro, e para isso deseja levá-la naquele restaurante chiquérrimo, mas fora o valor a ser desembolsado, precisa garantir que, ao chegar lá, vai ter mesa disponível. Então o que você faz? Horas antes, liga para o restaurante e faz uma reserva de uma mesa para dois. Assim, quando você chegar com sua companhia no restaurante na hora marcada, a mesa vai estar disponível, mesmo com o lugar estando lotado.

Trabalhar com variáveis em um programa funciona mais ou menos da mesma forma: ao se declarar uma variável em um programa, você ordena o seu computador fazer uma reserva de um endereço na memória RAM do seu computador, para que ele esteja disponível quando o seu programa precisar armazenar algum valor a ser utilizado pelo algoritmo.

E como funciona?

Em tese, a variável é sua, e portanto é você que tem o trabalho de dar um nome a ela, que pode ser qualquer um, desde que algumas regras sejam obedecidas. São elas:

- Não pode ser uma palavra reservada pelo sistema, como um comando já utilizado pela linguagem, por exemplo;
- Não pode ter acento;
- Não pode ter barras de espaço;
- Não pode ter cedilha (ç);
- Certos caracteres especiais são proibidos;
- Não pode ser igual ao nome de outra variável;
- Pode ter números, mas o nome não pode começar com um número.

Case Sensitive

Case Sensitive é o nome que damos às linguagens que fazem distinção entre letras maiúsculas e letras minúsculas. Por exemplo: uma variável chamada `nome` é diferente da variável `Nome`, que por sua vez é diferente da variável `NOME`, e assim por diante.

Dito isso, tome cuidado ao digitar o nome dos comandos. Observe se eles são digitados totalmente em letras minúsculas ou se possuem alguma letra maiúscula, pois isso fará a diferença entre seu código funcionar ou não.

Convenção de nomenclatura para variáveis em Python

Em tese, você pode nomear as variáveis do jeito que achar necessário, mas para facilitar a leitura e manutenção do código-fonte, algumas convenções costumam ser adotadas. Por exemplo, procure dar nomes simples às suas variáveis, sempre com letras totalmente minúsculas: `nome`, `idade`, `cidade`, etc...

Caso a sua variável seja um nome composto, separe os nomes por ***underscore*** (`_`). Por exemplo: `data_nascimento`, `idade_minima`, `valor_maximo`. Chamamos esse tipo de nomenclatura de **Snake Case**, e é o mais utilizado para a linguagem Python, diferentemente das outras linguagens de programação, que costumam utilizar o **Camel Case** para nomeação de variáveis. Evite também as letras maiúsculas no nome de suas variáveis.

Tipagem de variáveis no Python

Python é uma linguagem de tipagem dinâmica, ou seja, diferentemente de outras linguagens, como o Java ou o C, não informamos o tipo de dado que a variável irá receber na sua declaração. Você simplesmente informa o nome desejado para a sua variável, e atribui a ela um valor, em um processo que chamamos de **inicialização de variável**, e é assim que declaramos, tipamos, e inicializamos uma variável: com um único e simples comando.

Exemplo:

- `texto = 'Este é um texto'`
- `numero_inteiro = 12`
- `ponto_flutuante = 15.5`
- `valor_booleano = True`

Tipos de dados

Estes são os principais tipos de variáveis que trabalhamos em Python:

- **String:** é como são chamados textos pequenos, de até 255 caracteres. Geralmente são representados por palavras simples, compostas, frases, no máximo um parágrafo. Sempre estão dentro de aspas, sejam elas simples ou duplas.
- **Int:** são os números inteiros. Nunca são colocados dentro de aspas, e são representados apenas por números, nada mais. Nem vírgula, nem ponto, nem nenhum outro tipo de caractere.
- **Pontos flutuantes:** são os números decimais. São chamados assim pois os valores que os representam estão "flutuando" entre um valor e outro. As suas casas decimais são **SEMPRE** separados por pontos, e nunca por vírgulas.
- **Valores booleanos:** são valores binários de **VERDADEIRO** (`True`) ou **FALSO** (`False`), sempre com a inicial maiúscula. Uma variável desse tipo sempre vai receber um valor ou outro.

Atribuições de valores

É utilizado o sinal de igual (=) para atribuir um valor à uma variável. Lembre-se de que, na programação, o sinal de igual nunca significa igualdade, mas sim é usado para informar que algo recebeu um determinado valor.

Exibindo o valor de uma variável

Isso pode ser feito utilizando o comando `print()` , mas informando o nome da variável dentro do parênteses sem as aspas. Segue o código-fonte abaixo:

In [ ]:
```
# variável
texto = 'Este é o texto que deverá ser exibido na saída.'

# saída de dados
print(texto)
```

```
Este é o texto que deverá ser exibido na saída.
```

Exibindo o tipo de uma variável

Para identificar o tipo de uma variável, é possível usar o comando `type()` dentro de um `print()` em conjunto com uma variável. Segue o exemplo abaixo:

In [ ]:
```
# variáveis
nome = 'Alex'
idade = 39
altura = 1.72
```

```
rico = False

# saída dos tipos de variáveis
print(type(nome))
print(type(idade))
print(type(altura))
print(type(rico))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

Trocando o tipo de variável

Caso seja necessário, há a possibilidade de trocar o tipo de variável, chamando o tipo para o qual deseja trocar como uma função.

Exemplos:

Trocando um número para string:

In [ ]:
```
numero = 10
str(numero)
```

Out[ ]:
```
'10'
```

Trocando de int para float:

In [ ]:
```
numero = 10
float(numero)
```

Out[ ]:
```
10.0
```

Trocando de float para int:

In [ ]:
```
numero = 10.5
int(numero)
```

Out[ ]:
```
10
```

Concatenando string com variáveis

Concatenação é basicamente juntar valores. Em Python, há basicamente 4 formas de juntar strings com variáveis.

Forma 1: estilo do Java/JavaScript

Aqui se usa o sinal de **+** para concatenar valores, na mesma linha do Java e do JavaScript. Nessa forma, os dados do tipo int e float não podem ser imprimidos pelo comando, sendo necessário trocar o tipo dos números para string. Segue o código abaixo:

In [ ]:
```
# variáveis
nome = 'Alex'
```

```
idade = 39
altura = 1.72

# saída de dados
print('Olá, meu nome é ' + nome + ', tenho ' + str(idade) + ' anos, e tenho ' +
```

```
Olá, meu nome é Alex, tenho 39 anos, e tenho 1.72 metros de altura.
```

Forma 2: estilo C/C++/Portugol

Nesse estilo de concatenação, é usado a tradicional vírgula (,) para concatenação. Entretanto, diferente de outras linguagens, aqui ele já adiciona o espaço nas concatenações. Nesse caso, não há a necessidade de trocar o tipo das variáveis. Segue o exemplo do códdigo abaixo:

In [ ]:
```
# variáveis
nome = 'Alex'
idade = 39
altura = 1.72

# saída de dados
print('Olá, meu nome é ', nome, ', tenho ', idade, 'anos, e tenho ', altura, 'me
```

```
Olá, meu nome é  Alex , tenho  39 anos, e tenho  1.72 metros de altura.
```

Forma 3: String format

Nessa forma, é usado a função `format()` para concatenar. É um pouco mais avançado, e não é recomendado para iniciantes.

In [ ]:
```
# variáveis
nome = 'Alex'
idade = 39
altura = 1.72

# saída de dados
print('Olá, meu nome é {}, tenho {} anos, e tenho {} metros de altura.'.format(n
```

```
Olá, meu nome é Alex, tenho 39 anos, e tenho 1.72 metros de altura.
```

Forma 4: f-string

Essa é a forma mais usada pelos devs Python, pois deixa o código mais limpo e elegante. Se adiciona a letra **f** minúscula antes da string, e as variáveis são embutidas dentro das chaves, que por usa vez vão dentro das strings. **Obs:** essa será a forma mais utilizada durante o curso. Segue o exemplo abaixo:

In [ ]:
```
# variáveis
nome = 'Alex'
idade = 39
altura = 1.72

# saída de dados
print(f'Olá, meu nome é {nome}, tenho {idade} anos, e tenho {altura} metros de a
```

```
Olá, meu nome é Alex, tenho 39 anos, e tenho 1.72 metros de altura.
```

Eliminando quebra de linha

Diferente de outras linguagens, em Python você não precisa executar a famosa **quebra de linha** (`\n`) para exibir valores na tela. O problema é que nem sempre você deseja fazer essa quebra de linha, e quando isso acontece, esse recurso do Python se torna indesejado. Se for do seu desejo não fazer a quebra de linha entre dois `print()` , o algoritmo abaixo poderá ajudar a resolver esse problema, ao usar o comando `end=''` no final de uma string:

In [ ]:
```
print('O Sábio sabia ', end='')
print('que o sabiá sabia assobiar.')
```

```
O Sábio sabia que o sabiá sabia assobiar.
```

Alterando número de casas decimais de pontos flutuantes

Caso seja necessário, há uma forma de definir quantas casas decimais um valor flutuante pode ter. Para isso, siga o exemplo do código abaixo, que pega o valor de uma variável com várias casas decimais, e diminui para apenas duas:

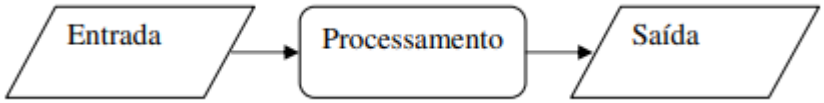
In [ ]:
```
valor = 1.233456789

# exibindo o valor com duas casas decimais
print(f'{valor:,.2f}')
```

```
1.23
```

Entrada de dados do Python

Como aprendemos em Informática Básica, qualquer sistema informático realiza 3 funções básicas em uma sequência específica:



Na aula de hoje, já aprendemos como fazer a parte da **saída de dados** em Python através da função `print()` . Mas todos os dados que passaram pela saída de dados foram produzidas pelo próprio algoritmo. E se o usuário quiser ele mesmo entrar com uma informação? Nesse caso, o programador não tem como saber qual valor ele irá informar para colocar no programa. É aí onde entra o comando de **entrada de dados** do Python. Para que a entrada de dados funcione, é importante que na execução do programa, o usuário receba uma informação, ou seja, um texto indicando o que deve ser feito, para que ele saiba que tipo de informação ele pode inserir no programa. Nas outras linguagens de programação, isso é feito inserindo um comando de saída de dados com a informação antes do comando de entrada de dados associado à variável que irá armazenar a informação digitada pelo usuário. Em Python, tudo isso pode ser feito pelo mesmo comando usando a função `input()` . Veja o exemplo à seguir:

In [ ]:
```
# variável é declarada e recebe um input por parte do usuário
nome = input('Informe o seu nome:')

# programa exibe o dado informado pelo usuário
print(nome)
```

```
Alex Machado
```

Alterando vírgula para ponto

Conforme visto anteriormente, o ponto é o que separa as casas decimais, e não a vírgula. Acontece que o usuário final não sabe disso, nem é obrigado a saber, e por isso, caso ele seja solicitado a informar um número decimal, ele inevitavelmente irá informá-lo separando as casas decimais com vírgula, o que fará nosso programa retornar um erro. Para evitar isso, podemos "converter" a vírgula digitada pelo usuário para o ponto, usando a função `replace()`, e depois convertendo para um `float`. Veja a seguir como usar:

In [ ]:
```
valor_decimal = input('Inform um número decimal: ')
valor_decimal = valor_decimal.replace(',', '.')
valor_decimal = float(valor_decimal)

print(valor_decimal)
```

```
1.25
```

Exercícios

---

1. Crie um arquivo **.py** e faça um algoritmo que receba do usuário o nome, idade e profissão, e imprima na tela essas informações.

In [ ]:
```
# TODO: jogue o código aqui
```