
Python do
ZERO
à Programação
Orientada a Objetos
Fernando Feltrin
Edição atualizada e ampliada.

Índice

Por quê programar? E por quê em Python?

No âmbito da tecnologia da informação, basicamente começamos a dividir seus nichos entre a parte física (hardware) e sua parte lógica (software), e dentro de cada uma delas existe uma infinidade de subdivisões, cada uma com suas particularidades e usabilidades diferentes.

O aspirante a profissional de T.I. pode escolher entre várias muitas áreas de atuação, e mesmo escolhendo um nicho bastante específico ainda assim há um mundo de conhecimento a ser explorado. Dentro da parte lógica um dos diferenciais é a área da programação, tanto pela sua complexidade quanto por sua vasta gama de possibilidades.

Sendo assim um dos diferenciais mais importantes do profissional de tecnologia moderno é o mesmo ter certa bagagem de conhecimento de programação. No ambito acadêmico existem diversos cursos, que vão de análise e desenvolvimento de sistemas até engenharia da computação, e da maneira como esses cursos são organizados você irá reparar que sempre haverá em sua grade curricular uma carga horária dedicada a programação. No Brasil a linguagem de programação mais popularmente utilizada nos cursos de tecnologia é C ou uma de suas vertentes, isso se dá pelo fato de C ser uma linguagem ainda hoje bastante popular e que pode servir de base para tantas outras.

Quando estamos falando especificamente da área da programação existe uma infinidade de linguagens de programação que foram sendo desenvolvidas ao longo do tempo para suprir a necessidade de criação de softwares que atendessem uma determinada demanda. Poderíamos dedicar um capítulo inteiro mostrando apenas as principais e suas características, mas ao invés disso vamos nos focar logo em Python.

Hoje com a chamada internet das coisas, data science, machine learning, além é claro da criação de softwares, jogos e sistemas, mais do que nunca foi preciso profissionais da área que soubessem programação. Python é uma linguagem idealizada e criada na década de 80 mas que se mostra hoje uma das mais modernas e promissoras, devido sua facilidade de aprendizado e sua capacidade de se adaptar a qualquer situação. Se você buscar qualquer comparativo de Python em relação a outras linguagens de programação garanto que em 95% dos casos Python sairá em vantagem.

Python pode ser a sua linguagem de programação definitiva, ou abrir muitas portas para aprender outras mais, já que aqui não existe uma real concorrência, a melhor linguagem sempre será aquela que irá se adaptar melhor ao programador e ao projeto a ser desenvolvido.

Sendo assim, independentemente se você já é programador de outra linguagem ou se você está começando do zero, espero que o conteúdo deste pequeno livro seja de grande valia para seu aprendizado dentro dessa área incrível.

Metodologia

Este material foi elaborado com uma metodologia autodidata, de forma que cada conceito será explicado de forma progressiva, sucinta e exemplificado em seguida.

Cada tópico terá seus exemplos e devida explicação, assim como sempre que necessário você terá o código na íntegra e em seguida sua 'engenharia reversa', explicando ponto a ponto o que está sendo feito e os porquês de cada argumento dentro do código.

Cada tópico terá ao menos um exemplo; Cada termo dedicado a programação terá destaque para que você o diferencie em meio ao texto ou explicações.

Desde já tenha em mente que aprender a programar requer atenção e mais do que isso, muita prática. Sendo assim, recomendo que sempre que possível pratique recriando os códigos de exemplo e não tenha medo que testar diferentes possibilidades em cima deles.

Dadas as considerações iniciais, mãos à obra!!!

1 – Introdução

Quando estamos iniciando nossa jornada de aprendizado de uma linguagem de programação é importante que tenhamos alguns conceitos bem claros já logo de início. O primeiro deles é que aprender uma linguagem de programação não é muito diferente do que aprender outro idioma falado, haverá uma sintaxe e uma sequência lógica de argumentos a se respeitar a fim de que seus comandos façam sentido e funcionem.

Com certeza quando você começou a aprender inglês na escola você começou pelo verbo **to be** e posteriormente foi incrementando com novos conceitos e vocabulário, até ter certo domínio sobre o mesmo. Em uma linguagem de programação não é muito diferente, há uma sintaxe, que é a maneira com que o interpretador irá reconhecer seus comandos, e há também uma lógica de programação a ser seguida uma vez que queremos através de linhas/blocos de código dar instruções ao computador e chegar a algum resultado.

Se você pesquisar, apenas por curiosidade, sobre as linguagens de programação você verá que outrora elas eram extremamente complexas, com uma curva de aprendizado longo e que por fim eram pouco eficientes ou de uso bastante restrito. As linguagens de alto nível, como Python, foram se modernizando de forma a que hoje é possível fazer muito com pouco, de forma descomplicada e com poucas linhas de código já estaremos criando programas e/ou fazendo a manutenção dos mesmos.

Se você realmente tem interesse por essa área arrisco dizer que você irá adorar Python e programação em geral. Então chega de enrolação e vamos começar... do começo.

Por quê Python?

Como já mencionei anteriormente, um dos grandes diferenciais do Python, que normalmente é o chamativo inicial por quem busca uma linguagem de programação, é sua facilidade, seu potencial de fazer mais com menos. Não desmerecendo outras linguagens, mas é fato que Python foi desenvolvida para ser descomplicada. E por quê fazer as coisas da forma mais difícil se existem ferramentas para torná-las mais fáceis?

Existe uma piada interna do pessoal que programa em Python que diz:

"- A vida é curta demais para programar em outra linguagem senão em Python."

Como exemplo veja três situações, um simples programa que exibe em tela a mensagem "Olá Mundo!!!" escrito em C, em JAVA, e porfim em Python.

Olá Mundo!!! em **C**

```c
#include<stdio.h>
int main (void)
{
printf("Ola Mundo!!!\n");
return 0;
}
```

Olá Mundo!!! em **JAVA**

```java
public class hello {
public static void main (String arg []){
   System.out.println("Olá Mundo!!!");
}
}
```

Olá Mundo!!! em **Python**

```python
print('Olá Mundo!!!')
```

Em todos os exemplos acima o retorno será uma mensagem exibida em tela para o usuário dizendo: **Olá Mundo!!!** (*Garanto que agora a piada fez sentido...)

De forma geral o primeiro grande destaque do Python frente a outras linguagens é sua capacidade de fazer mais com menos, e em todas as situações que conheço este padrão se repetirá, será muito mais fácil, mais rápido, com menos linhas de código, programar em Python.

Outro grande diferencial do Python em relação a outras linguagens de programação é o fato dela ser uma linguagem interpretada, em outras palavras, o ambiente de programação que você irá trabalhar tem capacidade de rodar o código em tempo real e de forma nativa, diferente de outras linguagens que tem que estar emulando uma série de parâmetros do sistema ou até mesmo compilando o código para que aí sim seja possível testá-lo.

Outro diferencial é a sua simplicidade sintática, posteriormente você aprenderá sobre a sintaxe Python, mas por hora apenas imagine que em programação (outras linguagens) muitas vezes temos uma grande idéia e ao codificá-la acabamos nos frustrando porque o código simplesmente não funciona, e em boa parte das vezes é por conta de um ponto ou vírgula que ficou sobrando ou faltando no código. Python por ser uma linguagem interpratada deveria sofrer ainda mais com esse problemas mas o que ocorre é ao contrário, o interpretador foca nos comandos e seus parâmetros, os pequenos erros de sintaxe não farão com que o código não funcione...

Por fim, outro grande diferencial do Python se comparado a outras linguagens de programação é que ela nativamente possui um núcleo capaz de trabalhar com uma

quantidade enorme de dados de forma bastante tranquila, o que a fez virar a queridinha de quem trabalha com data science, machine learning, blockchain, e outras tecnologias que trabalham e processam volumes enormes de dados.

Na verdade, eu poderia escrever outro livro só comparando Python com outras linguagens e mostrando suas vantagens, mas acho que por hora já é o suficiente para lhe deixar empolgado, e ao longo desse curso você irá descobrir um potencial muito grande em Python.

Um pouco de história

Em 1989, através do Instituto de Pesquisa Nacional para Matemática e Ciência da Computação, Guido Van Rossum publicava a primeira versão do Python. Derivada do C, a construção do Python se deu inicialmente para ser uma alternativa mais simples e produtiva do que o próprio C. Por fim em 1991 a linguagem Python ganhava sua "versão estável" e já funcional, começando a gerar também uma comunidade dedicada a aprimorá-la. Somente em 1994 foi lançada sua versão 1.0, ou seja, sua primeira versão oficial e não mais de testes, e de lá para cá houveram gigantecas melhorias na linguagem em si, seja em estrutura quanto em bibliotecas e plugins criadas pela comunidade para implementar novos recursos a mesma e a tornar ainda mais robusta.

Atualmente o Python está integrado em praticamente todas novas tecnologias, assim como é muito fácil implementá-la em sistemas "obsoletos". Grande parte das distribuições Linux possuem Python nativamente e seu reconhecimento já fez com que, por exemplo, virasse a linguagem padrão do curso de ciências da computação do MIT desde 2009.

Guido Van Rossum

Não posso deixar de falar, ao menos um pouco, sobre a mente por tras da criação do Python, este cara chamado Guido Van Rossum. Guido é um premiado matemático e programador que hoje se dedica ao seu emprego atual na Dropbox. Guido já trabalhou em grandes empresas no passado e tem um grande reconhecimento por estudantes e profissionais da computação.

Dentre outros projetos, Guido em 1991 lançou a primeira versão de sua própria linguagem de programação, o Python, que desde lá sofreu inúmeras melhorias tanto pelos seus desenvolvedores quanto pela comunidade e ainda hoje ele continua supervisionando o desenvolvimento da linguagem Python, tomando as decisões quando necessário.

A filosofia do Python

Python tem uma filosofia própria, ou seja, uma série de porquês que são responsáveis por Python ter sido criada e por não ser “só mais uma linguagem de programação”.

Por Tim Peters, um influente programador de Python

1. Bonito é melhor que feio.
2. Explícito é melhor que implícito.
3. Simples é melhor que complexo.
4. Complexo é melhor que complicado.
5. Plano é melhor que aglomerado.
6. Escasso é melhor que denso.
7. O que conta é a legibilidade.
8. Casos especiais não são especiais o bastante para quebrar as regras.
9. A natureza prática derruba a teórica.
10. Erros nunca deveriam passar silenciosamente.
11. a menos que explicitamente silenciasse.
12. Diante da ambigüidade, recuse a tentação de adivinhar.
13. Deveria haver um -- e preferivelmente só um -- modo óbvio para fazer as coisas.
14. Embora aquele modo possa não ser óbvio a menos que você seja holandês.
15. Agora é melhor que nunca.
*16. Embora nunca é freqüentemente melhor que *agora mesmo*.*
17. Se a implementação é difícil para explicar, isto é uma idéia ruim.
18. Se a implementação é fácil para explicar, pode ser uma idéia boa.
19. Namespaces são uma grande idéia -- façamos mais desses!

Essa filosofia é o que fez com que se agregasse uma comunidade enorme disposta a investir seu tempo em Python. Em suma, programar em Python deve ser simples, de fácil aprendizado, com código de fácil leitura, enxuto mas inteligível, capaz de se adaptar a qualquer necessidade.

Empresas que usam Python

Quando falamos de linguagens de programação, você já deve ter reparado que existem inúmeras delas, mas basicamente podemos dividí-las em duas grandes categorias: Linguagens específicas e/ou Linguagens Generalistas. Uma linguagem específica, como o próprio nome sugere, é aquela linguagem que foi projetada para atender a um determinado propósito fixo, como exemplo podemos citar o PHP e o HTML, que são linguagens específicas para web. Já linguagens generalistas são aquelas

que tem sua aplicabilidade em todo e qualquer propósito, e nem por isso ser inferior às específicas.

No caso do Python, ela é uma linguagem generalista bastante moderna, é possível criar qualquer tipo de sistema para qualquer propósito e plataforma a partir dela. Só para citar alguns exemplos, em Python é possível programar para qualquer sistema operacional, web, mobile, data science, machine learning, blockchain, etc... coisa que outras linguagens de forma nativa não são suficientes ou práticas para o programador, necessitando uma série de gambiarras para realizar sua codificação, tornando o processo mais difícil.

Como exemplo de aplicações que usam parcial ou totalmente Python podemos citar YouTube, Google, Instagram, Dropbox, Quora, Pinterest, Spotify, Reddit, Blender 3D, BitTorrent, etc... Apenas como curiosidade, em computação gráfica dependendo sua aplicação uma engine pode trabalhar com volumosos dados de informação, processamento e renderização, a Light and Magic, empresa subsidiária da Disney, que produz de maneira absurda animações e filmes com muita computação gráfica, usa de motores gráficos escritos e processados em Python devido sua performance.

O futuro da linguagem

De acordo com as estatísticas de sites especializados, Python é uma das linguagens com maior crescimento em relação às demais no mesmo período, isto se deve pela popularização que a linguagem recebeu após justamente grandes empresas declarar que a adotaram e comunidades gigantescas se formarem para explorar seu potencial. Em países mais desenvolvidos tecnologicamente até mesmo escolas de ensino fundamental estão adotando o ensino de programação em sua grade de disciplinas, boa parte delas, ensinando nativamente Python.

Por fim, podemos esperar para o futuro que a linguagem Python cresça exponencialmente, uma vez que novas áreas de atuação como data science e machine learning se popularizem ainda mais. Estudos indicam que para os próximos 10 anos cerca de um milhão de novas vagas surgirão demandando profissionais de tecnologia da área da programação, pode ter certeza que grande parcela desse público serão programadores com domínio em Python.

Python será o limite?

Esta é uma pergunta interessante de se fazer porque precisamos parar uns instantes e pensar no futuro. Raciocine que temos hoje um crescimento exponencial do

uso de machine learning, data science, internet das coisas, logo, para o futuro podemos esperar uma demanda cada vez maior de processamento de dados, o que não necessariamente signifique que será mais complexo desenvolver ferramentas para suprir tal demanda.

A versão 3 do Python é bastante robusta e consegue de forma natural já trabalhar com tais tecnologias. Devido a comunidade enorme que desenvolve para Python, podemos esperar que para o futuro haverão novas versões implementando novos recursos de forma natural. Será muito difícil vermos surgir outra linguagem “do zero” ou que tenha usabilidade parecida com Python.

Se você olhar para trás verá uma série de linguagens que foram descontinuadas com o tempo, mesmo seus desenvolvedores insistindo e injetando tempo em dinheiro em seu desenvolvimento elas não eram modernas o suficiente. Do meu ponto de vista não consigo ver um cenário do futuro ao qual Python não consiga se adaptar.

Entenda que na verdade não é uma questão de uma linguagem concorrer contra outra, na verdade independente de qual linguagem formos usar, precisamos de uma que seja capaz de se adaptar ao seu tempo e as nossas necessidades. Num futuro próximo nosso diferencial como profissionais da área de tecnologia será ter conhecimento sobre C#, Java ou Python. Garanto que você já sabe em qual delas estou apostando minhas fichas...

2 – Ambiente de programação

Na linguagem Python, e, não diferente das outras linguagens de programação, quando partimos do campo das idéias para a prática, para codificação/programação não basta que tenhamos um computador rodando seu sistema operacional nativo. É importante começarmos a raciocinar que, a partir do momento que estamos entrando na programação de um sistema, estamos trabalhando com seu backend, ou seja, com o que está por trás das cortinas, com aquilo que o usuário final não tem acesso. Para isto, existe uma gama enorme de softwares e ferramentas que nos irão auxiliar a criar nossos programas e levá-los ao frontend.

Linguagens de alto, baixo nível e de máquina

Seguindo o raciocínio lógico do tópico anterior, agora entramos nos conceitos de linguagens de alto e baixo nível e posteriormente a linguagem de máquina.

Quando estamos falando em linguagens de alto e baixo nível, estamos falando sobre o quão distante está a sintaxe do usuário. Para ficar mais claro, uma linguagem de alto nível é aquela mais próximo do usuário, que usa termos e conceitos normalmente vindos do inglês e que o usuário pode pegar qualquer bloco de código e o mesmo será legível e fácil de compreender. Em oposição ao conceito anterior, uma linguagem de baixo nível é aquela mais próxima da máquina, com instruções que fazem mais sentido ao interpretador do que ao usuário.

Quando estamos programando em Python estamos num ambiente de linguagem de alto nível, onde usaremos expressões em inglês e uma sintaxe fácil para por fim dar nossas instruções ao computador. Esta linguagem posteriormente será convertida em linguagem de baixo nível e por fim se tornará sequências de instruções para registradores e portas lógicas. Imagine que o comando que você digita para exibir um determinado texto em tela é convertido para um segundo código que o interpretador irá ler como bytecode, convertendo ele para uma linguagem de mais baixo nível chamada Assembly, que irá pegar tais instruções e converter para binário, para que por fim tais instruções virem sequências de chaveamento para portas lógicas do processador.

Nos primórdios da computação se convertiam algoritmos em sequências de cartões perfurados que iriam programar sequências de chaveamento em máquinas ainda valvuladas, com o surgimento dos transistores entramos na era da eletrônica digital onde foi possível miniaturizar os registradores, trabalhar com milhares deles de forma a realizar centenas de milhares de cálculos por segundo. Desenvolvemos linguagens de programação de alto nível para que justamente facilitássemos a leitura e

escrita de nossos códigos, porém a linguagem de máquina ainda é, e por muito tempo será, binário, ou seja, sequências de informações de zeros e uns que se convertem em pulsos ou ausência de pulsos elétricos nos transistores do processador.

Ambientes de desenvolvimento integrado

Entendidos os conceitos de linguagens de alto e baixo nível e de máquina, por fim vamos falar sobre os IDEs, sigla para ambiente de desenvolvimento integrado. Já rodando nosso sistema operacional temos a frontend do sistema, ou seja, a capa ao qual o usuário tem acesso aos recursos do mesmo. Para que possamos ter acesso aos bastidores do sistema e por fim programar em cima dele, temos softwares específicos para isto, os chamados IDE's. Nestes ambientes temos as ferramentas necessárias para tanto trabalhar a nível de código quanto para testar o funcionamento de nossos programas no sistema operacional.

As IDEs vieram para integrar todos softwares necessários para esse processo de programação em um único ambiente, já houveram épocas onde se programava separado de onde se compilava, separado de onde se debugava e por fim separado de onde se rodava o programa, entenda que as ides unificam todas camadas necessárias para que possamos nos concentrar em escrever nossos códigos e rodá-los ao final do processo. Entenda que é possível programar em qualquer editor de texto, como o próprio bloco de notas ou em um terminal, o uso de IDEs se dá pela facilidade de possuir todo um espectro de ferramentas dedicadas em um mesmo lugar.

Principais IDEs

Como mencionado no tópico anterior, as IDEs buscam unificar as ferramentas necessárias para facilitar a vida do programador, claro que é possível programar a partir de qualquer bloco de notas, mas visando ter um ambiente completo onde se possa usar diversas ferramentas, testar e compilar seu código, o uso de uma IDE é altamente recomendado.

Existem diversas IDEs disponíveis no mercado, mas basicamente aqui irei recomendar duas delas.

Pycharm – A IDE Pycharm desenvolvida e mantida pela JetBrains, é uma excelente opção no sentido de que possui versão completamente gratuita, é bastante intuitiva e fácil de aprender a usar seus recursos e por fim ela oferece um ambiente com suporte em tempo real ao uso de console/terminal próprio, sendo possível a qualquer momento executar testes nos seus blocos de código.

Disponível em: https://www.jetbrains.com/pycharm/

Anaconda – Já a suite Anaconda, também gratuita, conta com uma série de ferramentas que se destacam por suas aplicabilidades. Python é uma linguagem muito usada para data science, machine learning, e a suíte anaconda oferece softwares onde é possível trabalhar dentro dessas modalidades com ferramentas dedicadas a ela, além, é claro, do próprio ambiente de programação comum, que neste caso é o VisualStudioCode, e o Jupyter que é um terminal que roda via browser.

Disponível em: https://www.anaconda.com/download/

Importante salientar também que, é perfeitamente normal e possível programar direto em terminal, seja em linux, seja em terminais oferecidos pelas próprias IDEs, a escolha de usar um ou outro é de particularidade do programador. É possível inclusive usar um terminal junto ao próprio editor da IDE.

Como dito anteriormente, existem diversas IDEs e ferramentas que facilitarão sua vida como programador, a verdade é que o interessante mesmo você dedicar um tempinho a testar as duas e ver qual você se adapta melhor. Não existe uma IDE melhor que a outra, o que existem são ferramentas que se adaptam melhor as suas necessidades enquanto programador.

3 – Lógica de programação

Como já mencionei em um tópico anterior, uma linguagem de programação é uma linguagem como qualquer outra em essência, o diferencial é que na programação são meios que criamos para conseguir passar comandos a serem executados em um computador. Quando falamos sobre um determinado assunto, automaticamente em nossa língua falada nativa geramos uma sentença que possui uma certa sintaxe e lógica de argumento para que a outra pessoa entenda o que estamos querendo transmitir. Se não nos expressarmos de forma clara podemos não ser entendidos ou o pior, ser mal-entendidos.

Um computador é um mecanismo exato, ele não tem (ainda) a capacidade de discernimento e abstração que possuímos, logo, precisamos passar suas instruções de forma literal e ordenada, para que a execução de um processo seja correta.

Quando estudamos lógica de programação, estudamos métodos de criar sequências lógicas de instrução, para que possamos usar tais sequências para programar qualquer dispositivo para que realize uma determinada função. Essa sequência lógica recebe o nome de algoritmo.

Algoritmos

Todo estudante de computação no início de seu curso recebe essa aula, algoritmos. Algoritmos em suma nada mais é do que uma sequência de passos onde iremos passar uma determinada instrução a ser realizada. Imagine uma receita de bolo, onde há informação de quais ingredientes serão usados e um passo a passo de que ordem cada ingrediente será adicionado e misturado para que no final do processo o bolo dê certo, seja comestível e saboroso.

Um algoritmo é exatamente a mesma coisa, conforme temos que programar uma determinada operação, temos que passar as instruções passo a passo para que o interpretador consiga as executar e chegar ao fim de um processo.

Quando estudamos algoritmos basicamente temos três tipos básicos de algoritmo. Os que possuem uma entrada e geram uma saída, os que não possuem entrada mas geram saída e os que possuem entrada mas não geram saída.

Parece confuso mas calma, imagine que na sua receita de bolo você precise pegar ingrediente por ingrediente e misturar, para que o bolo fique pronto depois de assado (**neste caso temos entradas que geram uma saída.**). Em outro cenário imagine que você tem um sachê com a mistura já pronta de ingredientes, bastando apenas adicionar leite e colocar para assar (**neste caso, não temos as entradas mas temos a saída**). Por fim

imagine que você é a empresa que produz o sachê com os ingredientes já misturados e pré-prontos, você tem os ingredientes e produz a mistura, porém não é você que fará o bolo (**neste caso temos as entradas e nenhuma saída**).

Em nossos programas haverão situações onde iremos criar scripts que serão executados ou não de acordo com as entradas necessárias e as saídas esperadas para resolver algum tipo de problema computacional.

Sintaxe em Python

Você provavelmente não se lembra do seu processo de alfabetização, quando aprendeu do zero a escrever suas primeiras letras, sílabas, palavras até o momento em que interpretou sentenças inteiras de acordo com a lógica formal de escrita, e muito provavelmente hoje o faz de forma automática apenas transliterando seus pensamentos. Costumo dizer que aprender uma linguagem de programação é muito parecido, você literalmente estará aprendendo do zero um meio de "se comunicar" com o computador de forma a lhe passar instruções a serem executadas dentro de uma ordem e de uma lógica.

Toda linguagem de programação tem sua maneira de transliterar a lógica de um determinado algoritmo em uma linguagem característica que será interpretada "pelo computador", isto chamamos de sintaxe. Toda linguagem de programação tem sua sintaxe característica, o programador deve respeitá-la até porque o interpretador é uma camada de software o qual é "programado" para entender apenas o que está sob a sintaxe correta. Outro ponto importante é que uma vez que você domina a sintaxe da linguagem ao qual está trabalhando, ainda assim haverão erros em seus códigos devido a erros de lógica.

Não se preocupe porque a coisa mais comum, até mesmo para programadores experientes é a questão de vez em quando cometer algum pequeno erro de sintaxe, ou não conseguir transliterar seu algoritmo numa lógica correta.

A sintaxe na verdade será entendida ao longo deste livro, até porque cada tópico que será abordado, será algum tipo de instrução que estaremos aprendendo e haverá uma forma correta de codificar tal instrução para que finalmente seja interpretada.

Já erros de lógica são bastante comuns nesse meio e, felizmente ou infelizmente, uma coisa depende da outra. Raciocine que você tem uma idéia, por exemplo, de criar uma simples calculadora. De nada adianta você saber programar e não entender o funcionamento de uma calculadora, ou o contrário, você saber toda lógica de funcionamento de uma calculadora e não ter idéia de como transformar isto em código.

Portanto, iremos bater muito nessa tecla ao longo do livro, tentando sempre entender qual a lógica por trás do código e a codificação de seu algoritmo.

Para facilitar, nada melhor do que usarmos um exemplo prático de erro de sintaxe e de lógica respectivamente.

Supondo que queiramos exibir em tela a soma de dois números. 5 e 2, respectivamente.

*Esses exemplos usam de comandos que serão entendidos detalhadamente em capítulos posteriores, por hora, entenda que existe uma maneira certa de se passar informações para "a máquina", e essas informações precisam obrigatoriamente ser em uma linguagem que faça sentido para o interpretador da mesma.

Ex 1:

```
print('5' + '2')
```

O resultado será **52**, porque de acordo com a sintaxe do Python 3, tudo o que digitamos entre 'aspas' é uma **string** (texto), sendo assim o interpretador pegou o texto '5' e o texto '2' e, como são dois textos, os concatenou de acordo com o símbolo de soma. Em outras palavras, aqui o nosso objetivo de somar 5 com 2 não foi realizado com sucesso porque passamos a informação para "a máquina" de maneira errada. Um erro típico de lógica.

Ex 2:

```
print(5 + 2)
```

O resultado é **7**, já que o interpretador pegou os dois valores inteiros 5 e 2 e os somou de acordo com o símbolo **+**, neste caso um operador matemático de soma. Sendo assim, codificando da maneira correta o interpretador consegue realizar a operação necessária.

Ex 3:

```
Print(5) + 2 =
```

O interpretador irá acusar um erro de sintaxe nesta linha do código pois ele não reconhece **Print** (iniciado em maiúsculo) e também não entende quais são os dados a serem usados e seus operadores. Um erro típico de sintaxe, uma vez que não estamos passando as informações de forma que o interpretador consiga interpretá-las adequadamente.

Em suma, como dito anteriormente, temos que ter bem definida a ideia do que queremos criar, e programar de forma que o interpretador consiga receber e trabalhar com essas informações. Sempre que nos depararmos com a situação de escrever alguma linha ou bloco de código e nosso interpretador acusar algum erro, devemos revisar o código em busca de que tipo de erro foi esse (lógica ou sintaxe) e procurar corrigir no próprio código.

Palavras reservadas

Na linguagem Python existem uma série de palavras reservadas para o sistema, ou seja, são palavras chave que o interpretador busca e usa para receber instruções a partir delas. Para ficar mais claro vamos pegar como exemplo a palavra **print**, em Python **print** é um comando que serve para exibir em tela ou em console um determinado dado ou valor, sendo assim, é impossível criarmos uma variável com nome **print**, pois esta é uma palavra reservada ao sistema.

Ao todo são 31 palavras reservadas na sintaxe.

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Repare que todas palavras utilizadas são termos em inglês, como Python é uma linguagem de alto nível, ela usa uma linguagem bastante próxima do usuário, com conhecimento básico de inglês é possível traduzir e interpretar o que cada palavra reservada faz.

A medida que formos progredindo você irá automaticamente associar que determinadas "palavras" são comandos ou instruções internas a linguagem. Você precisa falar uma lingua que a máquina possa reconhecer, para que no final das contas suas instruções sejam entendidas, interpretadas e executadas.

Análise léxica

Quando estamos programando estamos colocando em prática o que planejamos anteriormente sob a idéia de um algoritmo. Algoritmos por si só são sequências de instruções que iremos codificar para serem interpretadas e executar uma determinada função. Uma das etapas internas do processo de programação é a chamada análise léxica.

Raciocine que todo e qualquer código deve seguir uma sequência lógica, um passo-a-passo a ser seguido em uma ordem definida, basicamente quando estamos programando dividimos cada passo de nosso código em uma linha nova/diferente. Sendo assim, temos que respeitar a ordem e a sequência lógica dos fatos para que eles sempre ocorram na ordem certa.

O interpretador segue fielmente cada linha e seu conteúdo, uma após a outra, uma vez que "ele" é uma camada de programa sem capacidade de raciocínio e interpretação como nós humanos. Sendo assim devemos criar cada linha ou bloco de código em uma sequência passo-a-passo que faça sendito para o interpretador.

Por fim, é importante ter em mente que o interpretador sempre irá ler e executar, de forma sequencial, linha após linha e o conteúdo de cada linha sempre da esquerda para direita. Por exemplo:

```
print('Seja bem vindo!')
print('Você é' + 3 * 'muito' + 'legal!')
```

Como mencionamos anteriormente, o interpretador sempre fará a leitura das linhas de cima para baixo. Ex: linha 1, linha 2, linha 3, etc... e o interpretador sempre irá ler o conteúdo da linha da esquerda para direita.

Nesse caso o retorno que o usuário terá é:
Seja bem vindo!
Você é muito muito muito legal.

Repare que pela sintaxe a linha 2 do código tem algumas particularidades, existem 3 **strings** e um operador mandando repetir 3 vezes uma delas, no caso 3 vezes **'muito'**, e como são **strings**, o símbolo de **+** está servindo apenas para concatená-las. Embora isso ainda não faça sentido para você, fique tranquilo pois iremos entender cada ponto de cada linha de código posteriormente.

Indentação

Python é uma linguagem de forte indentação, ou seja, para fácil sintaxe e leitura, suas linhas de código não precisam necessariamente de uma pontuação, mas de uma tabulação correta. Quando linhas/blocos de código são filhos de uma determinada função ou parâmetro, devemos organizá-los de forma a que sua tabulação siga um determinado padrão.

Diferente de outras linguagens de programação que o interpretador percorre cada sentença e busca uma pontuação para demarcar seu fim, em Python o interpretador usa uma indentação para conseguir ver a hierarquia das sentenças.

O interpretador de Python irá considerar linhas e blocos de código que estão situados na mesma tabulação (margem) como blocos filhos do mesmo objeto/parâmetro. Para ficar mais claro, vejamos dois exemplos, o primeiro com indentação errada e o segundo com a indentação correta:

Indentação errada:

```
variavel1 = input('Digite um número: ')
if variavel1 >= str(0):
print('Número Positivo.')
print(f'O número é: {variavel1}')
else:
```

```
print('Número Negativo')
print(f'O número é: {variavel1}')
```

Repare que neste bloco de código não sabemos claramente que linhas de código são filhas e nem de quem... e mais importante que isto, com indentação errada o interpretador não saberá quais linhas ou blocos de código são filhas de quem, não conseguindo executar e gerar o retorno esperado, gerando erro de sintaxe.

Indentação correta:

```
variavel1 = input('Digite um número: ')

    if variavel1 >= str(0):
        print('Número Positivo.')
        print(f'O número é: {variavel1}')
    else:
        print('Número Negativo')
        print(f'O número é: {variavel1}')
```

Agora repare no mesmo bloco de código, mas com as indentações corretas, podemos ver de acordo com as margens e espaçamentos quais linhas de código são filhas de quais comandos ou parâmetros.

O mesmo ocorre por parte do interpretador, de acordo com a tabulação usada, ele consegue perceber que inicialmente existe uma variavel **variavel1** declarada que pede que o usuário digite um número, também que existem duas estruturas condicionais **if** e **else**, e que dentro delas existem instruções de imprimir em tela o resultado de acordo com o que o usuário digitou anteriormente e foi atribuido a **variavel1**.

Se você está vindo de outra linguagem de programação como **C** ou uma de suas derivadas, você deve estar acostumado a sempre encerrar uma sentença com ponto e vírgula **;** Em Python não é necessário uma pontuação, mas se por ventura você inserir, ele simplesmente irá reconhecer que ali naquele ponto e vírgula se encerra uma instrução. Isto é ótimo porque você verá que você volta e meia cometerá algumas excessões usando vícios de programação de outras linguagens que em Python ao invés de gerarmos um erro, o interpretador saberá como lidar com essa exceção. Ex:

Código correto:

```
print('Olá Amigo')
print('Tudo bem?')
```

Código com vício de linguagem:

```
print('Olá Amigo');
print('Tudo bem?')
```

Código extrapolado:

```
print('Olá Amigo'); print('Tudo bem?')
```

Nos três casos o interpretador irá contornar o que for vício de linguagem e entenderá a sintaxe prevista, executando normalmente os comandos **print( ).**
O retorno será: **Olá Amigo**
Tudo bem?

4 – Estrutura básica de um programa

Enquanto em outras linguagens de programação você tem de criar toda uma estrutura básica para que realmente você possa começar a programar, Python já nos oferece praticamente tudo o que precisamos pré-carregado de forma que ao abrirmos uma IDE nossa única preocupação inicial é realmente programar.

Python é uma linguagem "batteries included", termo em inglês para (pilhas inclusas), ou seja, ele já vem com o necessário para seu funcionamento pronto para uso. Posteriormente iremos implementar novas bibliotecas de funcionalidades em nosso código, mas é realmente muito bom você ter um ambiente de programação realmente pronto para uso.

Que tal começarmos pelo programa mais básico do mundo. Escreva um programa que mostre em tela a mensagem "Olá Mundo". Fácil não?

Vamos ao exemplo:

```
print('Olá Mundo!!!')
```

Sim, por mais simples que pareça, isto é tudo o que você precisará escrever para que de fato, seja exibida a mensagem **Olá Mundo!!!** na tela do usuário.

Note que existe no início da sentença um **print( )**, que é uma palavra reservada ao sistema que tem a função de mostrar algo no console ou na tela do usuário, **print( )** sempre será seguido de **( )** parênteses, pois dentro deles estará o que chamamos de argumentos/parâmetros dessa função, neste caso, uma **string** (frase) **'Olá Mundo!!!'**, toda frase, para ser reconhecida como tal, deve estar entre aspas. Então fazendo o raciocínio lógico desta linha de código, chamamos a função **print( )** que recebe como argumento **'Olá Mundo!!!'**.

O retorno será: **Olá Mundo!!!**

Em outras linguagens de programação você teria de importar bibliotecas para que fosse reconhecido mouse, teclado, suporte a entradas e saída em tela, definir um escopo, criar um método que iria chamar uma determinada função, etc... etc... etc... Em Python basta já de início dar o comando que você quer para que ele já possa ser executado. O interpretador já possui pré-carregado todos recursos necessários para identificar uma função e seus métodos, assim como os tipos de dados básicos que usaremos e todos seus operadores.

5 – Tipos de dados

Independentemente da linguagem de programação que você está aprendendo, na computação em geral trabalhamos com dados, e os classificamos conforme nossa necessidade. Para ficar mais claro, raciocine que na programação precisamos separar os dados quanto ao seu tipo. Por exemplo uma **string**, que é o termo reservado para qualquer tipo de dado alfanumérico (qualquer tipo de palavra/texto que contenha letras e números).

Já quando vamos fazer qualquer operação aritmética precisamos tratar os números conforme seu tipo, por exemplo o número **8**, que para programação é um **int** (número inteiro), enquanto o número **8.2** é um **float** (número com casa decimal). O ponto que você precisa entender de imediato é que não podemos misturar tipos de dados diferentes em nossas operações, porque o interpretador não irá conseguir distinguir que tipo de operação você quer realizar uma vez que ele faz uma leitura léxica e "literal" dos dados.

Por exemplo: Podemos dizer que Maria tem 8 anos, e nesse contexto, para o interpretador, o **8** é uma **string**, é como qualquer outra palavra dessa mesma sentença. Já quando pegamos dois números para somá-los por exemplo, o interpretador espera que esses números sejam **int** ou **float**, mas nunca uma **string**.

Parece muito confuso de imediato, mas com os exemplos que iremos posteriormente abordar você irá de forma automática diferenciar os dados quanto ao seu tipo e seu uso correto.

Segue um pequeno exemplo dos tipos de dados mais comuns que usaremos em Python:

Tipo	Descrição	Exemplo
Int	Número real inteiro, sem casas decimais	12
Float	Número com casas decimais	12.8
Bool	Booleano / Binário (0 ou 1)	0 (ou 1)
String	Texto com qualquer caractere alfanumérico	'palavra' "marca d'água"
List	Listas(s)	[2, 'Pedro', 15.9]
Dict	Dicionário(s)	{'nome':'João da Silva', 'idade': 32}

Repare também que cada tipo de dado possui uma sintaxe própria para que o interpretador os reconheça como tal. Vamos criar algumas variáveis (que veremos no capítulo a seguir) apenas para atribuir diferentes tipos de dados, de forma que possamos finalmente visualizar como cada tipo de dados deve ser representado.

```
variavel_tipo_string = 'Conjunto de caracteres alfanuméricos'
variavel_tipo_int = 12      #número inteiro
variavel_tipo_float = 15.3      #número com casas decimais
variavel_tipo_bool = 1      #Booleano / Binário (0 ou 1)
variavel_tipo_lista = [1, 2, 'Paulo', 'Maria', True]
```

```
variavel_tipo_dicionario = {'nome':'Fernando', 'idade':31,}
variavel_tipo_tupla = ('Ana', 'Fernando', 3)
variavel_tipo_comentario = """Um comentário atribuído a uma
                           variável deixa de ser apenas
                           um comentário, vira frase!!!"""
```

Representação visual:

6 – Comentários

Desculpe a redundância, mas comentários dentro das linguagens de programação servem realmente para comentar determinados blocos de código, para criar anotações sobre o mesmo. É uma prática bastante comum a medida que implementamos novas funcionalidades em nosso código ir comentando-o também, para facilitar nosso entendimento quando revisarmos o mesmo. Essa prática também é bastante comum quando pegamos códigos de domínio público, normalmente o código virá com comentários do seu autor explicando os porquês de determinadas linhas de código.

A sintaxe para comentar nossos códigos em Python é bastante simples, basicamente existem duas maneiras de comentar o código, quando queremos fazer um comentário que não será maior do que uma linha usaremos o símbolo **#** e em seguida o devido comentário. Já quando precisamos fazer algum comentário mais elaborado, que irá ter mais de uma linha, usaremos **‘’’**aspas triplas antes de nosso comentário e depois dele para o terminar**’’’**.

A idéia de comentar determinados blocos de código é uma maneira do programador colocar anotações sobre determinadas linhas de código. Tenha em mente que o interpretador não lê o que o usuário determinou como comentário, tudo o que estiver após **#** ou entre **‘’’** o interpretador irá simplesmente ignorar.

Vamos ao exemplo:

```
nome = ‘Maria’
#Maria é a nova funcionária

print(‘Bem vinda Maria!!!’)
#acima está uma mensagem de boas vindas a ela.
```

Exemplo 2:

```
'''Este programa está sendo escrito
para que Maria, a nova funcionária,
comece a se interar com o sistema.'''
```

Neste exemplo acima existem comentários dos dois tipos, o interpretador irá fazer apenas a leitura da variável **nome** e irá executar o comando **print( )**, ignorando todo o resto.

Porém se por ventura você quiser fazer com que um comentário passe a ser lido em seu código o mesmo deverá ser associado a uma variável. Ex:

```
'''Exemplo de comentário
   este, inclusive, não será
   lido pelo interpretador'''
```

Comentário interno, não lido pelo interpretador e não visível pelo usuário. Apenas comentário para alguma linha/bloco de código.

```
comentario1 = '''Exemplo de comentário
                e agora sim será lido
                pelo interpretador!!!'''
print(comentario1)
```

Comentário que será exibido ao usuário, pois está associado a uma variável e o comando **print( )** está executando sua exibição.
O retorno será: **Exemplo de comentário**
e agora sim será lido
pelo interpretador!!!

Uma prática bastante comum é, enquanto testamos nossos blocos de código usar do artifício de comentário para isolar algum elemento, uma vez que este quando comentado passa a ser ignorado pelo interpretador.

```
var1 = 2019
var2 = 2020

soma = var1 + var2

print(soma)
```

Repare que o código acima por mais básico que seja possui duas variáveis **var1** e **var2**, uma operação de **soma** entre elas e um comando **print( )** que irá exibir em tela o resultado de **soma**. Se quisermos por exemplo, apenas a fim de testes, ignorar a operação **soma** declarada no código e realizar esta operação diretamente dentro do comando **print( )**, isto é perfeitamente possível. Bastando "comentar" a função **soma**, ela passa a ser ignorada pelo interpretador. Ex:

```
var1 = 2019
var2 = 2020
var3 = 2021

#soma = var1 + var2

print(var1 + var2)
```

O retorno em ambos os casos será: **4039**

Note que no segundo bloco de código, ao inserir o marcador **#** a frente da função **soma**, a transformamos em um simples comentário, ignorado pelo interpretador, para traze-la de volta a ativa basta "descomentar" a mesma.

7 – Variáveis / objetos

Uma variável basicamente é um espaço alocado na memória ao qual iremos armazenar um dado, valor ou informação. Imagine que você tem uma escrivaninha com várias gavetas, uma variável é uma dessas gavetas ao qual podemos guardar dentro dela qualquer coisa (qualquer tipo de dado) e ao mesmo tempo ter acesso fácil a este dado durante a execução de nosso programa.

Python é uma linguagem dinamicamente tipada, ou seja, quando trabalhamos com variáveis/objetos (itens aos quais iremos atribuir dados ou valores), podemos trabalhar livremente com qualquer tipo de dado e se necessário, também alterar o tipo de uma variável a qualquer momento.

Outra característica importante de salientar neste momento é que Python, assim como outras linguagens, ao longo do tempo foi sofrendo uma série de mudanças que trouxeram melhorias em sua forma de uso. Na data de publicação deste livro estamos usando a versão 3.7 da linguagem Python, onde se comparado com as versões anteriores da mesma, houve uma série de simplificações em relação a maneira como declaramos variáveis, definimos funções, iteramos dados. Por fim apenas raciocine que não iremos estar nos focando em sintaxe antiga ou que está por ser descontinuada, não há sentido em nos atermos a isto, todo e qualquer código que será usado neste livro respeitará a sintaxe mais atual.

Declarando uma variável

A declaração básica de uma variável/objeto sempre seguirá uma estrutura lógica onde, toda variável deve ter um nome (desde que não seja uma palavra reservada ao sistema) e algo atribuído a ela (qualquer tipo de dado ou valor).

Partindo diretamente para prática, vamos ver um exemplo de declaração de uma variável:

```
variavel1 = 11
```

Neste caso, inicialmente declaramos uma variável de nome **variavel1** que por sua vez está recebendo o valor **10** como valor (o símbolo de **=** aqui é utilizado para atribuir um valor a variável). No contexto de declaração de variáveis, o símbolo de igualdade não está comparando ou igualando os lados, mas está sendo usado para atribuir um dado ou valor a uma variável.

```
variavel1 = 11
print(variavel1)
```

Aqui, inicialmente apenas para fins de exemplo, na segunda linha do código usamos o comando **print( )** que dentro de seus "parênteses" está instanciando a variável que acabamos de criar, a execução dessa linha de código irá exibir em tela para o usuário o dado/valor que for conteúdo, que estiver atribuído a variável **variavel1**.
Neste caso o retorno será: **11**

Conforme você progredir em seus estudos de programação você irá notar que é comum possuirmos várias variáveis, na verdade, quantas forem necessárias em nosso programa. Não existe um limite, você pode usar à vontade quantas variáveis forem necessárias desde que respeite a sua sintaxe e que elas tenham um propósito no código.

Outro ponto importante é que quando atribuímos qualquer dado ou valor a uma variável o tipo de dado é implícito, ou seja, se você por exemplo atribuir a uma variável simplesmente um número **6**, o interpretador automaticamente identifica esse dado como um **int** (dado numérico do tipo inteiro). Se você inserir um ponto '.' seguido de outro número, **5** por exemplo, tornando esse número agora **6.5**, o interpretador automaticamente irá reconhecer esse mesmo dado agora como **float** (número de ponto flutuante).

O mesmo ocorre quando você abre aspas ' ' para digitar algum dado a ser atribuído a uma variável, automaticamente o interpretador passará a trabalhar com aquele dado o tratando como do tipo **string** (palavra, texto ou qualquer combinação alfanumérica de caracteres). Posteriormente iremos ver que também é possível declarar explicitamente o tipo de um dado e até convertê-lo de um tipo para outro.

Python também é uma linguagem **case sensitive**, ou seja, ela diferencia caracteres maiúsculos de minúsculos, logo, existem certas maneiras de declarar variáveis que são permitidas enquanto outras não, gerando conflito com o interpretador. A grosso modo podemos declarar variáveis usando qualquer letra minúscula e o símbolo "_" underline simples no lugar do espaço.

Exemplo 1: Declarando variáveis corretamente.

```
variavel1 = 'Ana'
variavel2 = 'Pedro'

variavel_1 = 'Ana'
variavel_2 = 'Pedro'
```

Ambos os modelos apresentam sintaxe correta, cabe ao usuário escolher qual modo ele considera mais fácil de identificar essa variável pelo nome.

Exemplo 2: Simulando erro, declarando uma variável de forma não permitida.

```
variavel 1 = 'String'
14 = 'Numero'
```

Nem **variavel 1** nem a **14** serão reconhecidas pelo interpretador como variáveis porque estão escritas de forma não reconhecida pela sintaxe. No primeiro exemplo o espaço entre variável e 1 gera conflito com o interpretador. No segundo exemplo, um número nunca pode ser usado como nome para uma variável.

Embora permitido, não é recomendável usar o nome de uma variável todo em letras maiúsculas, por exemplo:

```
NOME = 'Fernando'
```

*Se você está vindo de outras linguagens de programação para o Python, você deve conhecer o conceito de que quando se declara uma variável com letras maiúsculas ela se torna uma constante, uma variável imutável. Esse conceito não existe para o Python porque nele as variáveis são tratadas como objetos e sempre serão dinâmicas. Por fim, fique tranquilo que usar essa sintaxe é permitida em Python, não muito recomendável, mas você não terá problemas em seu código em função disso.

Outro ponto importante de destacar é que em função do padrão **case sensitive**, ao declarar duas variáveis iguais, uma com caracteres maiúsculos e outra com caracteres minúsculos serão interpretadas como duas variáveis diferentes. Ex:

```
NOME = 'Fernando'
nome = 'Rafael'
```

Também é permitido, mas não é recomendado o uso de palavras com acentos: Ex:

```
variável = 'Maria'
cômodos = 3
```

Por fim, raciocine que o interessante é você criar o hábito de criar seus códigos usando as formas mais comuns de se declarar variáveis, deixando apenas para casos especiais essas exceções.

Exemplos de nomenclatura de variáveis permitidos:

```
variavel = 'Ana'
variavel1 = 'Ana'
variavel_1 = 'Ana'
var_num_1 = 'Ana'
minhavariavel = 'Ana'
minha_variavel = 'Ana'
minhaVariavel = 'Ana'
```

Permitidos, mas não recomendados:

```
variável = 'Ana'
VARIAVEL = 'Ana'
Variavel = 'Ana'
```

Não permitidos, por poder gerar conflitos com o interpretador de sua IDE:

```
1991 = 'Ana'
minha variavel = 'Ana'
1variavel = 'Ana'
```

*Existirão situações onde um objeto será declarado com letra maiúscula inicial, mas neste caso ele não é uma variável qualquer e seu modo de uso será totalmente diferente. Abordaremos esse tema nos capítulos finais do livro.

Vale lembrar também que por convenção é recomendado criar nomes pequenos e que tenham alguma lógica para o programador, se você vai declarar uma variável para armazenar o valor de um número, faz mais sentido declarar uma variável **numero1** do que algo tipo **ag_23421_m_meuNumero**...

Por fim, vale lembrar que, como visto em um capítulo anterior, existem palavras que são reservadas ao sistema, dessa forma você não conseguirá as usar como nome de uma variável. Por exemplo **while**, que é uma chamada para uma estrutura condicional, sendo assim, é impossível usar **while** como nome de variável.

Declarando múltiplas variáveis

É possível, para deixar o código mais enxuto, declarar várias variáveis em uma linha, independente do tipo de dado a ser recebido, desde que se respeite a sintaxe correta de acordo com o tipo de dado. Por exemplo, o código abaixo:

```
nome = 'Maria'
idade = 32
sexo = 'F'
altura = 1.89
```

Pode ser agrupado em uma linha, da sequinte forma:

```
nome, idade, sexo, altura = 'Maria', 32, 'F', 1.89
```

A ordem será respeitada e serão atribuídos os valores na ordem aos quais foram citados. (primeira variável com primeiro atributo, segunda variável com segundo atributo, etc...)

Declarando múltiplas variáveis (de mesmo tipo)

Quando formos definir várias variáveis, mas que possuem o mesmo valor ou tipo de dado em comum, podemos fazer essa declaração de maneira resumida. Ex:

Método convencional:

```
num1 = 10
x = 10
a1 = 10
```

Método resumido:

```
num1 = x = a1 = 10
```

Repare que neste caso, em uma linha de código foram declaradas 3 variáveis, todas associadas com o valor **10**, o interpretador as reconhecerá como variáveis independentes, podendo o usuário fazer o uso de qualquer uma delas isoladamente a qualquer momento. Por exemplo:

```
num1 = x = a1 = 10
```

8 – Funções Básicas

Funções, falando de forma bastante básica, são linhas ou blocos de códigos aos quais executarão uma determinada ação em nosso código, inicialmente trabalharemos com as funções mais básicas que existem, responsáveis por exibir em tela uma determinada resposta e também responsáveis por interagir com o usuário. Funções podem receber parâmetros de execução (ou não), dependendo a necessidade, uma vez que esses parâmetros nada mais são do que as informações de como os dados deverão interagir internamente para realizar uma determinada função.

Pela sintaxe Python, "chamamos" uma função pelo seu nome, logo em seguida, entre **( )** parênteses podemos definir seus parâmetros, instanciar variáveis ou escrever linhas de código à vontade, desde que não nos esqueçamos que este bloco de código fará parte (será de propriedade) desta função...

Posteriormente trataremos das funções propriamente ditas, como criamos funções personalizadas que realizam uma determinada ação, por hora, para conseguirmos dar prosseguimento em nossos estudos, precisamos entender que existem uma série de funções pré-programadas, prontas para uso, com parâmetros internos que podem ser implícitos ou explícitos ao usuário.

As mais comuns delas, **print( )** e **input( )** respectivamente nos permitirão exibir em tela o resultado de uma determinada ação de um bloco de código, e interagir com o usuário de forma que ele consiga por meio do teclado inserir dados em nosso programa.

Função print( )

Quando estamos criando nossos programas, é comum que de acordo com as instruções que programamos, recebamos alguma saída, seja ela alguma mensagem ou até mesmo a realização de uma nova tarefa. Uma das saídas mais comuns é exibirmos, seja na tela para o usuário ou em console (em programas que não possuem uma interface gráfica), uma mensagem, para isto, na linguagem python usamos a função **print( )**.

Na verdade anteriormente já usamos ela enquanto estávamos exibindo em tela o conteúdo de uma variável, mas naquele capítulo esse conceito de fazer o uso de uma função estava lá apenas como exemplo e para obtermos algum retorno de nossos primeiros códigos, agora iremos de fato entender o mecanismo de funcionamento deste tipo de função.

Por exemplo:

```
print('Seja bem vindo!!!')
```

Repare na sintaxe: a função **print( )** tem como parâmetro (o que está dentro de parênteses) uma **string** com a mensagem **Seja bem vindo!!!** Todo parâmetro é delimitado por **( )** parênteses e toda **string** é demarcada por ‘ ‘ aspas para que o interpretador reconheça esse tipo de dado como tal.
O retorno dessa linha de código será: **Seja bem vindo!!!**

Função input( )

Em todo e qualquer programa é natural que haja interação do usuário com o mesmo, de modo que com algum dispositivo de entrada o usuário dê instruções ou adicione dados. Começando pelo básico, em nossos programas a maneira mais rudimentar de captar os dados do usuário será por intermédio da função **input( )**, por meio dela podemos pedir, por exemplo que o usuário digite um dado ou valor, que internamente será atribuído a uma variável. Ex:

```
nome = input('Digite o seu nome: ')
print('Bem Vindo', nome)
```

Inicialmente declaramos uma variável de nome **nome** que recebe como atributo a função **input( )** que por sua vez dentro tem uma mensagem para o usuário. Assim que o usuário digitar alguma coisa e pressionar a tecla ENTER, esse dado será atribuído a variável **nome**. Em seguida, a função **print( )** exibe em tela uma mensagem definida concatenada ao nome digitado pelo usuário e atribuído a variável **nome**.
O retorno será: **Bem Vindo Fernando**
*Supondo, é claro, que o usuário digitou Fernando.

Apenas um adendo, como parâmetro de nossa função **print( )** podemos instanciar múltiplos tipos de dados, inclusive um tipo de dado mais de uma vez, apenas como exemplo, aprimorando o código anterior, podemos adicionar mais de uma **string** ao mesmo exemplo. Ex:

```
nome = input('Digite o seu nome: ')
print('Bem Vindo', nome,'!!!')
```

O retorno será: **Bem Vindo Fernando !!!**

Explorando a função print( )

Como mencionado anteriormente, existem muitas formas permitidas de se “escrever” a mesma coisa, e também existe a questão de que a linguagem Python ao

longo dos anos foi sofrendo alterações e atualizações, mudando aos poucos sua sintaxe. O importante de se lembrar é que, entre versões se aprimoraram certos pontos da sintaxe visando facilitar a vida do programador, porém, para programadores que de longa data já usavam pontos de uma sintaxe antiga, ela não necessariamente deixou de funcionar a medida que a linguagem foi sendo atualizada.

Raciocine que muitos dos códigos que você verá internet a fora estarão no padrão Python 2, e eles funcionam perfeitamente no Python, então, se você está aprendendo realmente do zero por meio deste livro, fique tranquilo que você está aprendendo com base na versão mais atualizada da linguagem. Se você programava em Python 2 e agora está buscando se atualizar, fique tranquilo também porque a sintaxe que você usava não foi descontinuada, ela gradualmente deixará de ser usada pela comunidade até o ponto de poder ser removida do núcleo do Python, por hora, ambas sintaxes funcionam perfeitamente. Vamos ver isso na prática:

print( ) básico – Apenas exibindo o conteúdo de uma variável:

```
nome1 = 'Maria'
print(nome1)
```

Inicialmente declaramos uma variável de nome **nome1** que recebe como atributo **'Maria'**, uma **string**. Em seguida exibimos em tela o conteúdo atribuído a **nome1**.

O retorno será: **Maria**

print( ) básico – Pedindo ao usuário que dê entrada de algum dado:

```
nome1 = input('Digite o seu nome: ')
print(nome1)
```

Declaramos uma variável de nome **nome1** que recebe como atributo a função **input( )** que por sua vez pede ao usuário que digite alguma coisa. Quando o usuário digitar o que for solicitado e pressionar a tecla ENTER, este dado/valor será atribuído a **nome1**. Da mesma forma que antes, por meio da função **print( )** exibimos em tela o conteúdo de **num1**.

O retorno será: **Maria**

*supondo que o usuário digitou Maria.

print( ) intermediário – Usando máscaras de substituição (Sintaxe antiga):

```
nome1 = input('Digite o seu nome: ')
print('Seja bem vindo(a) %s' %(nome1))
```

Da mesma forma que fizemos no exemplo anterior, declaramos uma variável **nome1** e por meio da função **input( )** pedidos uma informação ao usuário. Agora, como parâmetro de nossa função **print( )** temos uma mensagem (**string**) que reserva dentro

de si, por meio do marcador **%s**, um espaço a ser substituído pelo valor existente em **nome1**. Supondo que o usuário digitou Fernando.

O retorno será: **Seja bem vindo(a) Fernando**

Porém existem formas mais sofisticadas de de realizar esse processo de interação, e isso se dá por o que chamamos de máscaras de substituição. Máscaras de substituição foram inseridas na sintaxe Python com o intuito de quebrar a limitação que existia de poder instanciar apenas um dado em nossa **string** parâmetro de nossa função **print( )**. Com o uso de máscaras de substituição e da função **.format( )** podemos inserir um ou mais de um dado/valor a ser substituído dentro de nossos parâmetros de nossa função **print( ).** Ex:

```
nome1 = input('Digite o seu nome: ')
print('Seja bem vindo(a) {} !!!'.format(nome1))
```

A máscara **{ }** reserva um espaço dentro da string a ser substituída pelo dado/valor atribuído a **nome1**.

O retorno será: **Seja bem vindo(a) Fernando !!!**

Fazendo o uso de máscaras de substituição, como dito anteriormente, podemos instanciar mais de um dado/valor/variável dentro dos parâmetros de nossa função **print( ).** Ex:

```
nome1 = input('Digite o seu nome: ')
msg1 = 'Por favor entre'
msg2 = 'Você é o primeiro a chegar.'
print('Seja bem vindo {}, {}, {}'.format(nome1, msg1, msg2))
```

Repare que dessa vez temos mais duas variáveis **msg1** e **msg2** respectivamente que possuem **strings** como atributos. Em nossa função **print( )** criamos uma **string** de mensagem de boas vindas e criamos 3 máscaras de substituição, de acordo com a ordem definida em **.format( )** substituiremos a primeira pelo nome digitado pelo usuário, a segunda e a terceira máscara pelas frases atribuídas a **msg1** e **msg2**. Dessa forma:

O retorno será: **Seja bem vindo Fernando, Por favor entre, Você é o primeiro a chegar.**

print( ) intermediário – Usando "**f strings**" (Sintaxe nova/atual):

Apenas dando um passo adiante, uma maneira mais moderna de se trabalhar com máscaras de substituição se dá por meio de **f strings**, a partir da versão 3 do Python se permite usar um simples parâmetro "**f**" antes de uma string para que assim o interpretador subentenda que ali haverão máscaras de substituição a serem trabalhadas, independentemente do tipo de dado. Dessa forma de facilitou ainda mais o uso de máscaras dentro de nossa função **print( )** e outras em geral.

Basicamente declaramos um parâmetro “**f**” antes de qualquer outro e podemos instanciar o que quisermos diretamente dentro das máscaras de substituição, desde o conteúdo de uma variável até mesmo operações e funções dentro de funções, mas começando pelo básico, veja o exemplo:

```
nome = input('Digite o seu nome: ')
ap = input('Digite o número do seu apartamento: ')
print(f'Seja bem vinda {nome}, moradora do ap nº {ap}')
```

Supondo que o usuário digitou respectivamente Maria e 33...

O retorno será: **Seja bem vinda Maria, moradora do ap nº 33**

Quando for necessário exibir uma mensagem muito grande, de mais de uma linha, uma forma de simplificar nosso código reduzindo o número de prints a serem executados é usar a quebra de linha dentro de um **print( )**, adicionando um **\n** frente ao texto que deverá estar posicionado em uma nova linha. Ex:

```
nome = 'João'
dia_vencimento = 10
valor_fatura = 149.90

print(f'Olá, caro {nome},\n A sua última fatura com vencimento
em {dia_vencimento} de janeiro,\n no valor de R${valor_fatura}
está próxima do vencimento.\n Favor pagar até o prazo para
evitar multas.')
```

Repare que existe um comando **print( )** com uma sentença enorme, que irá gerar 4 linhas de texto de retorno, combinando o uso de máscaras para sua composição.
O retorno será:

Olá, caro João,
A sua última fatura com vencimento em 10 de janeiro,
no valor de R$149.9 está próxima do vencimento.
Favor pagar até o prazo para evitar multas.

O que você deve ter em mente, e começar a praticar, é que em nossos programas, mesmo os mais básicos, sempre haverá meios de interagir com o usuário, seja exibindo certo conteúdo para o mesmo, seja interagindo diretamente com ele de forma que ele forneça ou manipule os dados do programa.

Lembre-se que um algoritmo pode ter entradas, para execução de uma ou mais funções e gerar uma ou mais saídas. Estas entradas pode ser um dado importado, um link ou um arquivo instanciado, ou qualquer coisa que esteja sendo inserida por um dispositivo de entrada do computador, como o teclado que o usuário digita ou o mouse interagindo com alguma interface gráfica.

Interação entre variáveis

Agora entendidos os conceitos básicos de como fazer o uso de variáveis, como exibir seu conteúdo em tela por meio da função print( ) e até mesmo interagir com o usuário via função input( ), hora de voltarmos a trabalhar os conceitos de variáveis, aprofundando um pouco mais sobre o que pode ser possível fazer a partir delas.

A partir do momento que declaramos variáveis e atribuímos valores a elas, podemos fazer a interação entre elas (interação entre seus atributos), por exemplo:

```
num1 = 10
num2 = 5.2
soma = num1 + num2
print(soma)
print(f'O resultado é {soma}')
```

Inicialmente criamos uma variável **num1** que recebe como atributo **10** e uma segunda variável **num2** que recebe como atributo **5.2**. Na sequência criamos uma variável **soma** que faz a soma entre **num1** e **num2** realizando a operação instanciando as próprias variáveis. O resultado da soma será guardado em **soma**. A partir daí podemos simplesmente exibir em tela via função **print( )** o valor de **soma**, assim como podemos criar uma mensagem mais elaborada usando máscara de substituição. Dessa forma...

O retorno será: **15.2**

Seguindo com o que aprendemos no capítulo anterior, podemos melhorar ainda mais esse código realizando este tipo de operação básica diretamente dentro da máscara de substituição. Ex:

```
num1 = 10
num2 = 5.2
print(f'O resultado é {num1 + num2}')
```

O retorno será: **15.2**

Como mencionado anteriormente, o fato da linguagem Python ser dinamicamente tipada nos permite a qualquer momento alterar o valor e o tipo de uma variável. Por exemplo, a variável **a** que antes tinha o valor **10** (**int**) podemos a qualquer momento alterar para **'Maria'** (**string**). Por exemplo:

```
a = 10
print(a)
```

O resultado será: **10**

```
a = 10
a = 'Maria'
print(a)
```

O resultado será: **Maria.**

A ordem de leitura por parte do interpretador o último dado/valor atribuído a variável **a** foi **'Maria'**. Como explicado nos capítulos iniciais, a leitura léxica desse código respeita a ordem das linhas de código, ao alterarmos o dado/valor de uma variável, o interpretador irá considerar a última linha de código a qual se fazia referência a essa variável e seu último dado/valor atribuído.

Por fim, quando estamos usando variaveis dentro de alguns tipos de operadores podemos temporariamente convertê-los para um tipo de dado, ou deixar mais explícito para o interpretador que tipo de dado estamos trabalhando para que não haja conflito. Por exemplo:

```
num1 = 5
num2 = 8.2
soma = int(num1) + int(num2)
print(soma)
```

O resultado será: **13** (sem casas decimais, porque definimos na expressão de **soma** que **num1** e **num2** serão tratados como **int**, número inteiro, sem casas decimais).

Existe a possibilidade também de já deixar especificado de que tipo de dado estamos falando quando o declaramos em uma variável. Por exemplo:

```
num1 = int(5)
num2 = float(8.2)
soma = num1 + num2
print(soma)
```

A regra geral diz que qualquer operação entre um **int** e um **float** resultará em **float.**
O returno será **13.2**

Como você deve estar reparando, a sintaxe em Python é flexível, no sentido de que haverão várias maneiras de codificar a mesma coisa, deixando a escolha por parte do usuário. Apenas aproveitando o exemplo acima, duas maneiras de realizar a mesma operação de soma dos valores atribuídos as respectivas variáveis.

```
num1 = 5
num2 = 8.2
soma = int(num1) + int(num2)

# Mesmo que:
num1 = int(5)
num2 = float(8.2)
soma = num1 + num2
```

Por fim, é possível "transformar" de um tipo numérico para outro apenas alterando a sua declaração. Por exemplo:

```
num1 = int(5)
num2 = float(5)
```

```
print(num1)
print(num2)
```

O retorno será: **5**

5.0

Note que no segundo retorno, o valor **5** foi declarado e atribuído a **num2** como do tipo **float**, sendo assim, ao usar esse valor, mesmo inicialmente ele não ter sido declarado com sua casa decimal, a mesma aparecerá nas operações e resultados.

```
num1 = int(5)
num2 = int(8.2)
soma = num1 + num2
print(soma)
```

Mesmo exemplo do anterior, mas agora já especificamos que o valor de **num2**, apesar de ser um número com casa decimal, deve ser tratado como inteiro, e sendo assim:
O Retorno será: **13**

E apenas concluindo o raciocínio, podemos aprimorar nosso código realizando as operações de forma mais eficiente, por meio de **f strings**. Ex:

```
num1 = int(5)
num2 = int(8.2)
print(f'O resultado da soma é: {num1 + num2}')
```

O Retorno será: **13**

Conversão de tipos de dados

É importante entendermos que alguns tipos de dados podem ser “misturados” enquanto outros não, quando os atribuímos a nossas variáveis e tentamos realizar interações entre as memsa. Porém existem recursos para elucidar tanto ao usuário quanto ao interpretador que tipo de dado é em questão, assim como podemos, conforme nossa necessidade, convertê-los de um tipo para outro.

A forma mais básica de se verificar o tipo de um dado é por meio da função **type( ).** Ex:

```
numero = 5
print(type(numero))
```

O resultado será: **<class 'int'>**

O que será exibido em tela é o tipo de dado, neste caso, um **int**.

Declarando a mesma variável, mas agora atribuindo **5** entre aspas, pela sintaxe, 5, mesmo sendo um número, será lido e interpretado pelo interpretador como uma **string**. Ex:

```
numero = '5'
print(type(numero))
```
O resultado será: **<class 'str'>**

Repare que agora, respeitando a sintaxe, **'5'** passa a ser uma **string**, um "texto".

Da mesma forma, sempre respeitando a sintaxe, podemos verificar o tipo de qualquer dado para nos certificarmos de seu tipo e presumir que funções podemos exercer sobre ele. Ex:

```
numero = [5]
print(type(numero))
```
O retorno será: **<class 'list'>**

```
numero = {5}
print(type(numero))
```
O retorno será: **<class 'set'>**

*Lembre-se que a conotação de chaves **{ }** para um tipo de dado normalmente faz dele um dicionário, nesse exemplo acima o tipo de dado retornado foi 'set' e não 'dict' em

Seguindo esta lógica e, respeitando o tipo de dado, podemos evitar erros de interpretação fazendo com que todo dado ou valor atribuído a uma variável já seja identificado como tal. Ex:

```
frase1 = str('Raquel tem 15 anos')

print(type(frase1))
```
Neste caso antes mesmo de atribuir um dado ou valor a variável **frase1** já especificamos que todo dado contido nela é do tipo **string**. Executando o comando **print(type(frase))** o retorno será: **<class 'str'>**

De acordo com o tipo de dados certas operações serão diferentes quanto ao seu contexto, por exemplo tendo duas frases atribuídas às suas respectivas variáveis, podemos usar o operador **+** para concatená-las (como são textos, soma de textos não existe, mas sim a junção entre eles). Ex:

```
frase1 = str('Raquel tem 15 anos, ')
frase2 = str('de verdade')

print(frase1 + frase2)
```
O resultado será: **Raquel tem 15 anos, de verdade.**

Já o mesmo não irá ocorrer se misturarmos os tipos de dados, por exemplo:

```
frase1 = str('Raquel tem ')
frase2 = int(15)
frase3 = 'de verdade'
```

```
print(frase1 + frase2 + frase3)
```

O retorno será um erro de sintaxe, pois estamos tentando juntar diferentes tipos de dados.

Corrigindo o exemplo anterior, usando as 3 variáveis como de mesmo tipo o comando **print** será executado normalmente.

```
frase1 = str('Raquel tem ')
frase2 = str(15)
frase3 = ' de verdade'

print(frase1 + frase2 + frase3)
```

O retorno será: **Raquel tem 15 de verdade.**

Apenas por curiosidade, repare que o código apresentado nesse último exemplo não necessariamente está usando f strings porque a maneira mais prática de o executar é instanciando diretamente as variáveis como parâmetro em nossa função print( ). Já que podemos optar por diferentes opções de sintaxe, podemos perfeitamente fazer o uso da qual considerarmos mais prática.

```
print(frase1 + frase2 + frase3)
# Mesmo que:
print(f'{frase1 + frase2 + frase3}')
```

Nesse exemplo em particular o uso de f strings está aumentando nosso código em alguns caracteres desnecessariamente.

Em suma, sempre preste muita atenção quanto ao tipo de dado e sua respectiva sintaxe, **5** é um **int** enquanto **‘5’** é uma **string**. Se necessário, converta-os para o tipo de dado correto para evitar erros de interpretação de seu código.

9 – Operadores

Operadores de Atribuição

Em programação trabalharemos com variáveis/objetos que nada mais são do que espaços alocados na memória onde iremos armazenar dados, para durante a execução de nosso programa, fazer o uso deles. Esses dados, independente do tipo, podem receber uma nomenclatura personalizada e particular que nos permitirá ao longo do código os referenciar ou incorporar dependendo a situação. Para atribuir um determinado dado/valor a uma variável teremos um operador que fará esse processo.

A atribuição padrão de um dado para uma variável, pela sintaxe do Python, é feita através do operador **=,** repare que o uso do símbolo de igual (**=**) usado uma vez tem a função de atribuidor, já quando realmente queremos usar o simbolo de igual para igualar operandos, usaremos ele duplicado (**==**). Por exemplo:

```
salario = 955
```

Nesta linha de código temos declaramos a variável **salario** que recebe como valor **955**, esse valor nesse caso é fixo, e sempre que referenciarmos a variável salario o interpretador usará seu valor atribuído, **955**.

Uma vez que temos um valor atribuído a uma variável podemos também realizar operações que a referenciem, por exemplo:

```
salario = 955
aumento1 = 27

print(salario + aumento1)
```

O resultado será **982**, porque o interpretador pegou os valores das variáveis **salario** e **aumento1** e os somou.

Por fim, também é possível fazer a atualização do valor de uma variável, por exemplo:

```
mensalidade = 229
mensalidade = 229 + 10

print(mensalidade)
```

O resultado será **239**, pois o último valor atribuido a variavel **mensalidade** era **229 + 10**.

Aproveitando o tópico, outra possibilidade que temos, já vimos anteriormente, e na verdade trabalharemos muito com ela, é a de solicitar que o usuário digite algum dado ou algum valor que será atribuido a variável, podendo assim, por meio do operador de atribuição, atualizar o dado ou valor declarado inicialmente, por exemplo:

```
nome = 'sem nome'
idade = 0

nome = input('Por favor, digite o seu nome: ')
idade = input('Digite a sua idade: ')

print(nome, idade)
```

Inicialmente as variáveis **nome** e **idade** tinham valores padrão pré-definidos, ao executar esse programa será solicitado que o usuário digite esses dados. Supondo que o usuário digitou **Fernando**, a partir deste momento a variável **nome** passa a ter como valor **Fernando**. Na sequência o usuário quando questionado sobre sua idade irá digitar números, supondo que digitou **33**, a variável idade a partir deste momento passa a ter como atribuição **33**. Internamente ocorre a atualização dessa variável para esses novos dados/valores atribuídos.

O retorno será: **Fernando 33**

Atribuições especiais

Atribuição Aditiva:

```
variavel1 = 4
variavel1 = variavel1 + 5

# Mesmo que:
variavel1 += 5

print(variavel1)
```

Com esse comando o usuário está acrescentando **5** ao valor de **variavel1** que inicialmente era **4**. Sendo 4 + 5:
O resultado será **9**.

Atribuição Subtrativa:

```
variavel1 = 4
variavel1 = variavel1 - 3

# Mesmo que:
variavel1 -= 3

print(variavel1)
```

Nesse caso, o usuário está subtraindo 3 de **variavel1**. Sendo 4 – 3:
O resultado será 1.

Atribuição Multiplicativa:

```
variavel1 = 4
variavel1 = variavel1 * 2

# Mesmo que:
variavel1 *= 2

print(variavel1)
```

Nesse caso, o usuário está multiplicando o valor de **variavel1** por 2. Logo 4 x 2:
O resultado será: **8**

Atribuição Divisiva:

```
variavel1 = 4
variavel1 = variavel1 / 4

# Mesmo que:
variavel1 /= 4

print(variavel1)
```

Nesse caso, o usuário está dividindo o valor de **variavel1** por 4. Sendo 4 / 4.
O resultado será: **1**

Módulo de (ou resto da divisão de):

```
variavel1 = 4
variavel1 = variavel1 % 4

# Mesmo que:
variavel1 %= 4

print(variavel1)
```

Será mostrado apenas o resto da divisão de **variavel1** por **4**.
O resultado será: **0**

Exponenciação:

```
variavel1 = 4
variavel1 = variavel1 ** 8

# Mesmo que:
variavel1 **= 8

print(variavel1)
```

Nesse caso, o valor de a será multiplicado 8 vezes por ele mesmo. Como a valia **4** inicialmente, a exponenciação será (4*4*4*4*4*4*4*4).
O resultado será: **65536**

Divisão Inteira:

```
variavel1 = 512
variavel1 = variavel1 // 512

# Mesmo que:
variavel1 //= 256

print(variavel1)
```

Neste caso a divisão retornará um número inteiro (ou arredondado). Ex: 512/256.
O resultado será: **2**

Operadores aritméticos

Os operadores aritméticos, como o nome sugere, são aqueles que usaremos para realizar operações matemáticas em nossos blocos de código. O Python por padrão já vem com bibliotecas pré-alocadas que nos permitem a qualquer momento fazer operações matemáticas simples como soma, subtração, multiplicação e divisão. Para operações de maior complexidade também é possível importar bibliotecas externas que irão implementar tais funções. Por hora, vamos começar do início, entendendo quais são os operadores que usaremos para realizarmos pequenas operações matemáticas.

Operador	Função
+	Realiza a soma de dois números
-	Realiza a subtração de dois números
*****	Realiza a multiplicação de dois números
/	Realiza a divisão de dois números

Como mencionei anteriormente, a biblioteca que nos permite realizar tais operações já vem carregada quando iniciamos nosso IDE, nos permitindo a qualquer momento realizar os cálculos básicos que forem necessários. Por exemplo:

Soma:

```
print(5 + 7)
```

O resultado será: **12**

Subtração:

```
print(12 - 3)
```

O resultado será: **9**

Multiplicação:

```
print(5 * 7)
```

O resultado será: **35**

Divisão:

```
print(120 / 6)
```

O resultado será: **20**

Operações com mais de 3 operandos:

```
print(5 + 2 * 7)
```

O resultado será **19** porque pela regra matemática primeiro se fazem as multiplicações e divisões para depois efetuar as somas ou subtrações, logo 2 x 7 são 14 que somados a 5 se tornam 19.

Operações dentro de operações:

```
print((5 + 2) * 7)
```

O resultado será 49 porque inicialmente é realizada a operação dentro dos parênteses (5 + 2) que resulta 7 e aí sim este valor é multiplicado por 7 fora dos parênteses.

Exponenciação:

```
print(3 ** 5) #3 elevado a 5ª potência
```

O resultado será **243**, ou seja, 3 x 3 x 3 x 3 x 3.

Outra operação possível e a de fazer uma divisão que retorne um número inteiro, "arredondado", através do operador **//**. Ex:

```
print(9.4 // 3)
```

O resultado será **3.0**, um valor arredondado.

Por fim também é possível obter somente o resto de uma divisão fazendo o uso do operador **%**. Por exemplo:

```
print(10 % 3)
```

O resultado será **1**, porque 10 divididos por 3 são 9 e seu resto é 1.

Apenas como exemplo, para encerrar este tópico, é importante você raciocinar que os exemplos que dei acima poderiam ser executados diretamente no console/terminal de sua IDE, mas claro que podemos usar tais operadores dentro de nossos blocos de código, inclusive atribuindo valores numéricos a variáveis e realizando operações entre elas. Por exemplo:

```
numero1 = 12
numero2 = 3
```

```
print(numero1 + numero2)

# Mesmo que:
print('O resultado da soma é:', numero1 + numero2)

# Que pode ser aprimorado para:
print(f'O resultado da soma é: {numero1 + numero2}')
```

O resultado será **15**, uma vez que **numero1** tem como valor atribuído 12, e **numero2** tem como valor atribuido 3. Somando as duas variáveis chegamos ao valor 15.

Exemplos com os demais operadores:

```
x = 5
y = 8
z = 13.2

print(x + y)
print(x - y)
print(x ** z)
print(z // y)
print(z / y)
```

Os resultados serão: **13**
-3
1684240309.400895
1.0
1.65

Operadores Lógicos

Operadores lógicos seguem a mesma base lógica dos operadores relacionais, inclusive nos retornando **True** ou **False**, mas com o diferencial de suportar expressões lógicas de maior complexidade.

Por exemplo, se executarmos diretamente no console a expressão **7 != 3** teremos o valor **True** (7 diferente de 3, verdadeiro), mas se executarmos por exemplo **7 != 3 and 2 > 3** teremos como retorno **False** (7 é diferente de 3, mas 2 não é maior que 3, e pela tabela verdade isso já caracteriza **False**).

```
print(7 != 3)
```

O retorno será: **True**

Afinal, 7 é diferente de 3.

```
print(7 != 3 and 2 > 3)
```

O retorno será: **False**

7 é diferente de 3 (Verdadeiro) e (and) 2 é maior que 3 (Falso).

Tabela verdade

Tabela verdade **AND** (E)

Independente de quais expressões forem usadas, se os resultados forem:

V E V = **V**
V E F = **F**
F E V = **F**
F E F = **F**

No caso do exemplo anterior, **7 era diferente de 3** (V) mas **2 não era maior que 3** (F), o retorno foi **False**.

Neste tipo de tabela verdade, bastando **uma** das proposições ser Falsa para que invalide todas as outras Verdadeiras. Ex:

V e V e V e V e V e V = **V**
V e F e V e V e V e V = **F**

Em python o operador **"and"** é um operador lógico (assim como os aritméticos) e pela sequência lógica em que o interpretador trabalha, a expressão é lida da seguinte forma:

```
7 != 3 and 2 > 3
True and False
False
# V e F = F
```

Analisando estas estruturas lógicas estamos tentando relacionar se **7 é diferente de 3** (Verdadeiro) e se **2 é maior que 3** (Falso), logo pela tabela verdade Verdadeiro e Falso resultará **False**.

Mesma lógica para operações mais complexas, e sempre respeitando a tabela verdade.

```
7 != 3 and 3 > 1 and 6 == 6 and 8 >= 9
True and True and True and False
False
```

7 diferente de 3 (V) E **3 maior que 1** (V) E **6 igual a 6** (V) E **8 maior ou igual a 9** (F).
É o mesmo que **True and True and True and False.**
Retornando **False** porque uma operação False já invalida todas as outras verdadeiras.

Tabela Verdade **OR** (OU)

Neste tipo de tabela verdade, mesmo tendo uma proposição Falsa, ela não invalida a Verdadeira. Ex:

V e V = **V**
V e F = **V**
F e V = **V**
F e F = **F**

Independente do número de proposições, bastando ter uma delas verdadeira já valida a expressão inteira.

V e V e V e V e V e V e V e V = **V**
F e F e F e F e F e F e F e F = **F**
F e F e F e F e F E F e V e F = **V**
F e F e F e F e F e F e F e F = **F**

Tabela Verdade **XOR** (OU Exclusivo/um ou outro)

Os dois do mesmo tipo de proposição são falsos, e nenhum é falso também.

V e V = **F**
V e F = **V**
F e V = **V**
F e F = **F**

Tabela de Operador de Negação (unário)

not True = F
O mesmo que dizer: Se não é verdadeiro então é falso.
not False = V
O mesmo que dizer: Se não é falso então é verdadeiro.

Também visto como:

not 0 = True
O mesmo que dizer: Se não for zero / Se é diferente de zero então é verdadeiro.
not 1 = False
O mesmo que dizer: Se não for um (ou qualquer valor) então é falso.

Bit-a-bit

O interpretador também pode fazer uma comparação bit-a-bit da seguinte forma:

AND Bit-a-bit

3 = 11 (3 em binário)
2 = 10 (2 em binário)
_ = 10

OR bit-a-bit

3 = 11
2 = 10
_ = 11

XOR bit-a-bit

3 = 11
2 = 10
_ = 01

Por fim, vamos ver um exemplo prático do uso de operadores lógicos, para que faça mais sentido.

```
saldo = 1000
salario = 4000
despesas = 2967

meta = saldo > 0 and salario - despesas >= 0.2 * salario
```

Analisando a variável **meta**: Ela verifica se **saldo** é maior que zero e se **salario** menos **despezas** é maior ou igual a 20% do **salário**.

O retorno será **True** porque o **saldo** era maior que zero e o valor de **salario** menos as **despesas** era maior ou igual a 20% do **salário**. (todas proposições foram verdadeiras).

Operadores de membro

Ainda dentro de operadores podemos fazer consulta dentro de uma lista obtendo a confirmação (**True**) ou a negação (**False**). Por Exemplo:

```
lista = [1, 2, 3, 'Ana', 'Maria']
```

```
print(2 in lista)
```
O retorno será: **True**

Lembrando que uma lista é definida por **[ ]** e seus valores podem ser de qualquer tipo, desde que separados por vírgula.

Ao executar o comando **2 in lista**, você está perguntando ao interpretador: **"2"** é membro desta lista? Se for (em qualquer posição) o retorno será **True.**

Também é possível fazer a negação lógica, por exemplo:

```
lista = [1, 2, 3, 'Ana', 'Maria']

print('Maria' not in lista)
```
O Retorno será **False.** 'Maria' não está na lista? A resposta foi **False** porque **'Maria'** está na lista.

Operadores relacionais

Operadores relacionais basicamente são aqueles que fazem a comparação de dois ou mais operandos, possuem uma sintaxe própria que também deve ser respeitada para que não haja conflito com o interpretador.

>	-	Maior que
>=	-	Maior ou igual a
<	-	Menor que
<=	-	Menor ou igual a
==	-	Igual a
!=	-	Diferente de

O retorno obtido no uso desses operadores será Verdadeiro (**True**) ou Falso (F**alse**).

Usando como referência o console, operando diretamente nele, ou por meio de nossa função **print( ),** sem declarar variáveis, podemos fazer alguns experimentos.

```
print(3 > 4)
#(3 é maior que 4?)
```
O resultado será **False.** 3 não é maior que 4.

```
print(7 >= 3)
#(7 é maior ou igual a 3?)
```
O resultado será **True.** 7 é maior ou igual a 3, neste caso, maior que 3.

```
print(3 >= 3)
#(3 é maior ou igual a 3?)
```

O resultado será **True**. 3 não é maior, mas é igual a 3.

Operadores usando variáveis

```
x = 2
y = 7
z = 5

print(x > z)
```

O retorno será **False.** Porque **x** (2) não é maior que **z** (5).

```
x = 2
y = 7
z = 5

print(z <= y)
```

O retorno será **True**. Porque **z** (5) não é igual, mas menor que **y** (7).

```
x = 2
y = 7
z = 5

print(y != x)
```

O retorno será **True** porque **y** (7) e diferente de **x** (2).

Operadores usando condicionais

```
if (2 > 1):
        print('2 é maior que 1')
```

Repare que esse bloco de código se iniciou com um **if**, ou seja, com uma condicional, **se** dois for maior do que um, então seria executado a linha de código abaixo, que exibe uma mensagem para o usuário: **2 é maior que 1**.

Mesmo exemplo usando valores atribuídos a variáveis e aplicando estruturas condicionais (que veremos em detalhe no capítulo seguinte):

```
num1 = 2
num2 = 1
```

```
if num1 > num2:
    print('2 é maior que 1')
```

O retorno será: **2 é maior que 1**

Operadores de identidade

Seguindo a mesma lógica dos outros operadores, podemos confirmar se diferentes objetos tem o mesmo dado ou valor atribuído. Por exemplo:

```
aluguel = 250
energia = 250
agua = 65

print(aluguel is energia)
```

O retorno será **True** porque os valores atribuídos são os mesmos (nesse caso, **250**).

10 – Estruturas condicionais

Quando aprendemos sobre lógica de programação e algoritmos, era fundamental entendermos que toda ação tem uma reação (mesmo que apenas interna ao sistema), dessa forma, conforme transliterávamos idéias para código, a coisa mais comum era nos depararmos com tomadas de decisão, que iriam influenciar os rumos da execução de nosso programa.

Muitos tipos de programas se baseiam em metodologias de estruturas condicionais, são programadas todas possíveis tomadas de decisão que o usuário pode ter e o programa executa e retorna certos aspectos conforme o usuário vai aderindo a certas opções.

Lembre-se das suas aulas de algoritmos, digamos que, apenas por exemplo o algoritmo **ir_ate_o_mercado** está sendo executado, e em determinada seção do mesmo existam as opções: **SE** estiver chovendo vá pela rua nº1, **SE NÃO** estiver chovendo, continue na rua nº2. Esta é uma tomada de decisão onde o usuário irá aderir a um rumo ou outro, mudando as vezes totalmente a execução do programa, desde que essas possibilidades estajam programadas. Não existe como o usuário tomar uma decisão que não está condicionada ao código, logo, todas possíveis tomadas de decisão dever ser programadas de forma lógica e responsiva.

Ifs, elifs e elses

Uma das partes mais legais de programação, sem sombra de dúvidas, é quando começamos a lidar com estruturas condicionais. Uma coisa é você ter um programa linear, que apenas executa uma tarefa após a outra, sem grandes interações e desfecho sempre linear, como um script passo-a-passo. Já outra coisa é você colocar condições, onde de acordo com as variáveis o programa pode tomar um rumo ou outro.

Como sempre, começando pelo básico, em Python a sintaxe para trabalhar com condicionais é bastante simples se comparado a outras linguagens de programação, basicamente temos os comandos **if** (se), **elif** (o mesmo que **else if** / mas se) e **else** (se não) e os usaremos de acordo com nosso algoritmo demandar tomadas de decisão.

A lógica de execução sempre se dará dessa forma, o interpretador estará executando o código linha por linha até que ele encontrará uma das palavras reservadas mencionadas anteriormente que sinaliza que naquele ponto existe uma tomada de decisão, de acordo com a decisão que o usuário indicar, ou de acordo com a validação de algum parâmetro, o código executará uma instrução, ou não executará nada, ignorando esta condição e pulando para o bloco de código seguinte.

Partindo pra prática:

```
a = 33
b = 34
c = 35

if b > a:
    print('b é MAIOR que a')
```

Declaradas três variáveis **a**, **b** e **c** com seus respectivos valores já atribuídos na linha abaixo existe a expressão **if**, uma tomada de decisão, sempre um **if** será seguido de uma instrução, que se for verdadeira, irá executar um bloco de instrução indentado a ela. Neste caso, se **b** for maior que **a** será executado o comando **print( )**.
O retorno será: **b é MAIOR que a**

*Caso o valor atribuído a **b** fosse menor que **a**, o interpretador simplesmente iria pular para o próximo bloco de código.

```
a = 33
b = 33
c = 35

if b > a:
    print('b é MAIOR que a')
elif b == a:
    print('b é IGUAL a a')
```

Repare que agora além da condicional **if** existe uma nova condicional declarada, o **elif**. Seguindo o método do interpretador, primeiro ele irá verificar se a condição de **if** é verdadeira, como não é, ele irá pular para esta segunda condicional. Por convenção da segunda condicional em diante se usa **elif**, porém se você usar **if** repetidas vezes, não há problema algum. Seguindo com o código, a segunda condicional coloca como instrução que se **b** for igual a **a**, e repare que nesse caso é, será executado o comando **print**.
O retorno será: **b é IGUAL a a**

```
a = 33
b = 1
c = 608

if b > a:
    print('b é MAIOR que a')
elif b == a:
    print('b é IGUAL a a')
else:
    print('b é MENOR que a')
```

Por fim, o comando **else** funciona como última condicional, ou uma condicional que é acionada quando nenhuma das condições anteriores do código for verdadeira, **else** pode ter um bloco de código próprio porém note que ele não precisa de nenhuma instrução, já que seu propósito é justamente apenas mostrar que nenhuma condicional (e sua instrução) anterior foi válida. Ex:

```python
var1 = 18
var2 = 2
var3 = 'Maria'
var4 = 4

if var2 > var1:
    print('A segunda variável é maior que a primeira')
elif var2 == 500:
    print('A segunda variável vale 500')
elif var3 == var2:
    print('A variavel 3 tem o mesmo valor da variavel 2')
elif var4 is str('4'):
    print('A variavel 4 não é do tipo string')
else:
    print('Nenhuma condição é verdadeira')
```

*Como comentamos rapidamente lá no início do livro, sobre algoritmos, estes podem ter uma saída ou não, aqui a saída é mostrar ao usuário esta mensagem de erro, porém se este fosse um código interno não haveria a necessidade dessa mensagem como retorno, supondo que essas condições simplesmente não fossem válidas o interpretador iria pular para o próximo bloco de código executando o que viesse a seguir.

Por fim, revisando os códigos anteriores, declaramos várias variáveis, e partir delas colocamos suas respectivas condições, onde de acordo com a validação destas condições, será impresso na tela uma mensagem. Importante entendermos também que o interpretador lê a instrução de uma condicional e se esta for verdadeira, ele irá executar o bloco de código e encerrar seu processo ali, pulando a verificação das outras condicionais.

Em suma, o interpretador irá verificar a primeira condicional, se esta for falsa, irá verificar a segunda e assim por diante até encontrar uma verdadeira, ali será executado o bloco de código indentado a ela e se encerrará o processo.

O legal é que como python é uma linguagem de programação dinamicamente tipada, você pode brincar a vontade alterando o valor das variáveis para verificar que tipo de mensagem aparece no seu terminal. Também é possível criar cadeias de tomada de decisão com inúmeras alternativas, mas sempre lembrando que pela sequência lógica em que o interpretador faz sua leitura é linha-a-linha, quando uma condição for verdadeira o algoritmo encerra a tomada de decisão naquele ponto.

And e Or dentro de condicionais

Também é possível combinar o uso de operadores **and** e **or** para elaborar condicionais mais complexas (de duas ou mais condições válidas). Por exemplo:

```python
a = 33
b = 34
c = 35

if a > b and c > a:
  print('a é maior que b e c é maior que a')
```

Se **a** for maior que **b** e **c** for maior que **a:**
O retorno será: **a é maior que b e c é maior que a**

```python
a = 33
b = 34
c = 35

if a > b or a > c:
  print('a é a variavel maior')
```

Se **a** for maior que **b** ou **a** for maior que **c.**
O retorno será: **a é a variavel maior**

Outro exemplo:

```python
nota = int(input('Informe a nota: '))

if nota >= 9:
   print('Parabéns, quadro de honra')
elif nota >= 7:
   print('Aprovado')
elif nota >= 5:
   print('Recuperação')
else:
   print('Reprovado')
```

Primeiro foi declarada uma variável de nome **nota**, onde será solicitado ao usuário que a atribua um valor, após o usuário digitar uma nota e apertar ENTER, o interpretador fará a leitura do mesmo e de acordo com as condições irá imprimir a mensagem adequada a situação.
Se **nota** for maior ou igual a **9**:
O retorno será: **Parabéns, quadro de honra**
Se **nota** for maior ou igual a **7**:
O retorno será: **Aprovado**
Se **nota** for maior ou igual a **5**:
O retorno será: **Recuperação**
Se **nota** não corresponder a nenhuma das proposições anteriores:
O retorno será: **Reprovado**

Condicionais dentro de condicionais

Outra prática comum é criar cadeias de estruturas condicionais, ou seja, blocos de código com condicionais dentro de condicionais. Python permite este uso e tudo funcionará perfeitamente desde que usada a sintaxe e indentação correta. Ex:

```
var1 = 0
var2 = int(input('Digite um número: '))

if var2 > var1:
    print('Numero maior que ZERO')
    if var2 == 1:
        print('O número digitado foi 1')
    elif var2 == 2:
        print('O número digitado foi 2')
    elif var2 == 3:
        print('O número digitado foi 3')
    else:
        print('O número digitado é maior que 3')

else:
    print('Número inválido')
```

Repare que foram criadas 2 variáveis **var1** e **var2**, a primeira já com o valor atribuído **0** e a segunda será um valor que o usuário digitar conforme solicitado pela mensagem, conertido para **int**. Em seguida foi colocada uma estrutura condicional onde se o valor de **var2** for maior do que **var1**, será executado o comando **print** e em seguida, dentro dessa condicional que já foi validada, existe uma segunda estrutura condicional, que com seus respectivos **ifs**, **elifs** e **elses** irá verificar que número é o valor de **var2** e assim irá apresentar a respectiva mensagem. Supondo que o usuário digitou 3:
O retorno será: **Numero maior que ZERO**
O número digitado foi 2

Fora dessa cadeia de condicionais (repare na indentação), ainda existe uma condicional **else** para caso o usuário digite um número inválido (um número negativo ou um caracter que não é um número).

Simulando switch/case

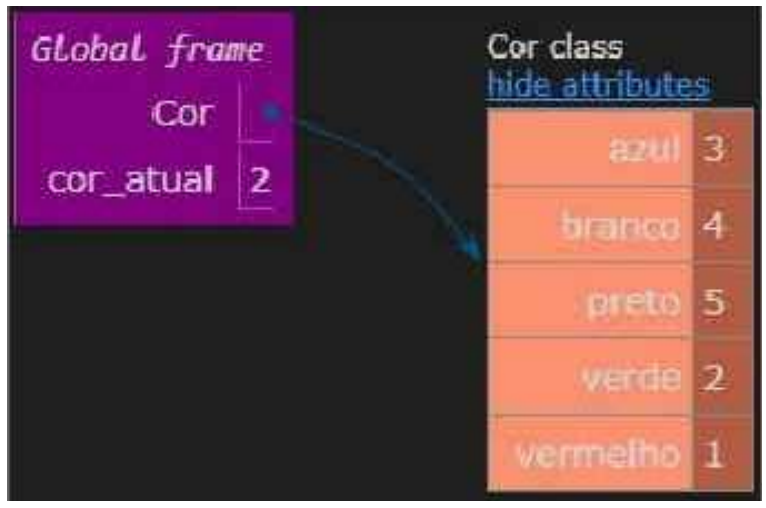
Quem está familiarizado com outras linguagens de programação está acostumado a usar a função **Swich** para que, de acordo com a necessidade, sejam tomadas certas decisões. Em Python nativamente não temos a função **switch**, porém temos como simular sua funcionalidade através de uma função onde definiremos uma variável com dicionário dentro, e como um dicionário trabalha com a lógica de **chave:valor,** podemos simular de acordo com uma **opção:umaopção**. Para ficar mais claro vamos ao código:

```python
class Cor:
  vermelho = 1
  verde = 2
  azul = 3
  branco = 4
  preto = 5

# Mude a cor para testar
cor_atual = 2

if cor_atual == Cor.vermelho:
  print("Vermelho")
elif cor_atual == Cor.verde:
  print("Verde")
elif cor_atual == Cor.azul:
  print("Azul")
elif cor_atual == Cor.branco:
  print("Branco")
elif cor_atual == Cor.preto:
  print("Preto")
else:
  print("Desconhecido")
```

Representação visual:



11 - Estruturas de repetição

While

Python tem dois tipos de comandos para executar comandos em loop (executar repetidas vezes uma instrução) o **while** e o **for**, inicialmente vamos entender como funciona o **while**. While do inglês, enquanto, ou seja, enquanto uma determinada condição for válida, a ação continuará sendo repetida. Por exemplo:

```
a = 1

while a < 8:
  print(a)
  a += 1
```

Declarada a variável **a**, de valor inicial **1** (pode ser qualquer valor, incluzive zero) colocamos a condição de que, enquanto o valor de **a** for menor que **8**, imprime o valor de a e acrescente (some) 1, repetidamente.
O retorno será: **1**
2
3
4
5
6
7
Repare que isto é um loop, ou seja, a cada ação o bloco de código salva seu último estado e repete a instrução, até atingir a condição proposta.

Outra possibilidade é de que durante uma execução de **while**, podemos programar um **break** (comando que para a execução de um determinado bloco de código ou instrução) que acontece se determinada condição for atingida. Normalmente o uso de **break** se dá quando colocamos mais de uma condição que, se a instrução do código atingir qualquer uma dessas condições (uma delas) ele para sua execução para que não entre em um loop infinito de repetições. Por exemplo:

```
a = 1

while a < 10:
  print(a)
  a += 1
  if a == 4:
     break
```

Enquanto a variável **a** for menor que **10**, continue imprimindo ela e acrescentando **1** ao seu valor. Mas se em algum momento ela for igual a **4**, pare a repetição. Como explicado

anteriormente, existem duas condições, se a execução do código chegar em uma delas, ele já dá por encerrada sua execução. Neste caso, se em algum momento o valor de **a** for **4** ou for um número maior que **10** ele para sua execução.

O resultado será:**1**
2
3
4

For

O comando **for** será muito utilizado quando quizermos trabalhar com um laço de repetição onde conhecemos os seus limites, ou seja, quando temos um objeto em forma de lista ou dicionário e queremos que uma variável percorra cada elemento dessa lista/dicionário interagindo com o mesmo. Ex:

```
compras = ['Arroz', 'Feijão', 'Carne', 'Pão']

for i in compras:
  print(i)
```

O retorno será: **Arroz**
Feijão
Carne
Pão

Note que inicialmente declaramos uma variável em forma de lista de nome **compras,** ele recebe 4 elementos do tipo **string** em sua composição. Em seguida declaramos o laço **for** e uma variável temporária de nome **i** (você pode usar o nome que quiser, e essa será uma variável temporária, apenas instanciada nesse laço / nesse bloco de código) que para cada execução dentro de **compras**, irá imprimir o próprio valor. Em outras palavras, a primeira vez que **i** entra nessa lista ela faz a leitura do elemento indexado na posição **0** e o imprime, encerrado o laço essa variável **i** agora entra novamente nessa lista e faz a leitura e exibição do elemento da posição **1** e assim por diante, até o último elemento encontrado nessa lista.

Outro uso bastante comum do **for** é quando sabemos o tamanho de um determinado intervalo, o número de elementos de uma lista, etc... e usamos seu método **in range** para que seja explorado todo esse intervalo. Ex:

```
for x in range(0, 6):
    print(f'Número {x}')
```

Repare que já de início existe o comando **for**, seguido de uma variável temporária **x**, logo em seguida está o comando **in range**, que basicamente define um intervalo a percorrer (de 0 até 6). Por fim há o comando **print** sobre a variável **x**.
O retorno será: **Número 0**
Número 1

Número 2
Número 3
Número 4
Número 5

*Note que a contagem dos elementos deste intervalo foi de **1** a **5**, **6** já está fora do range, serve apenas como orientação para o interpretador de que ali é o fim deste intervalo. Em Python não é feita a leitura deste último dígito indexado, o interpretador irá identificar que o limite máximo desse intervalo é 6, sendo **5** seu último elemento.

Quando estamos trabalhando com um intervalo há a possibilidade de declararmos apenas um valor como parâmetro, o interpretador o usará como orientação para o fimd e um intervalo. Ex:

```
for x in range(6):
    print(f'Número {x}')
```

O retorno será: **Número 0**
Número 1
Número 2
Número 3
Número 4
Número 5

Outro exemplo comum é quando já temos uma lista de elementos e queremos a percorrer e exibir seu conteúdo.

```
lista = ['Pedro', 255, 'Leticia']

for n in nomes:
    print(n)
```

Repare que existe uma lista inicial, com 3 dados inclusos, já quando executamos o comando **for** ele internamente irá percorrer todos valores contidos na lista nomes e incluir os mesmos na variável **n**, independentemente do tipo de dado que cada elemento da lista é, por fim o comando **print** foi dado em cima da variável **n** para que seja exibido ao usuário cada elemento dessa lista.

O resultado será: **Pedro**
255
Letícia

Em Python o laço **for** pode nativamente trabalhar como uma espécie de condicional, sendo assim podemos usar o comando **else** para incluir novas instruções no mesmo. Ex:

```
nomes = ['Pedro', 'João', 'Leticia',]

for laco in nomes:
    print(laco)
```

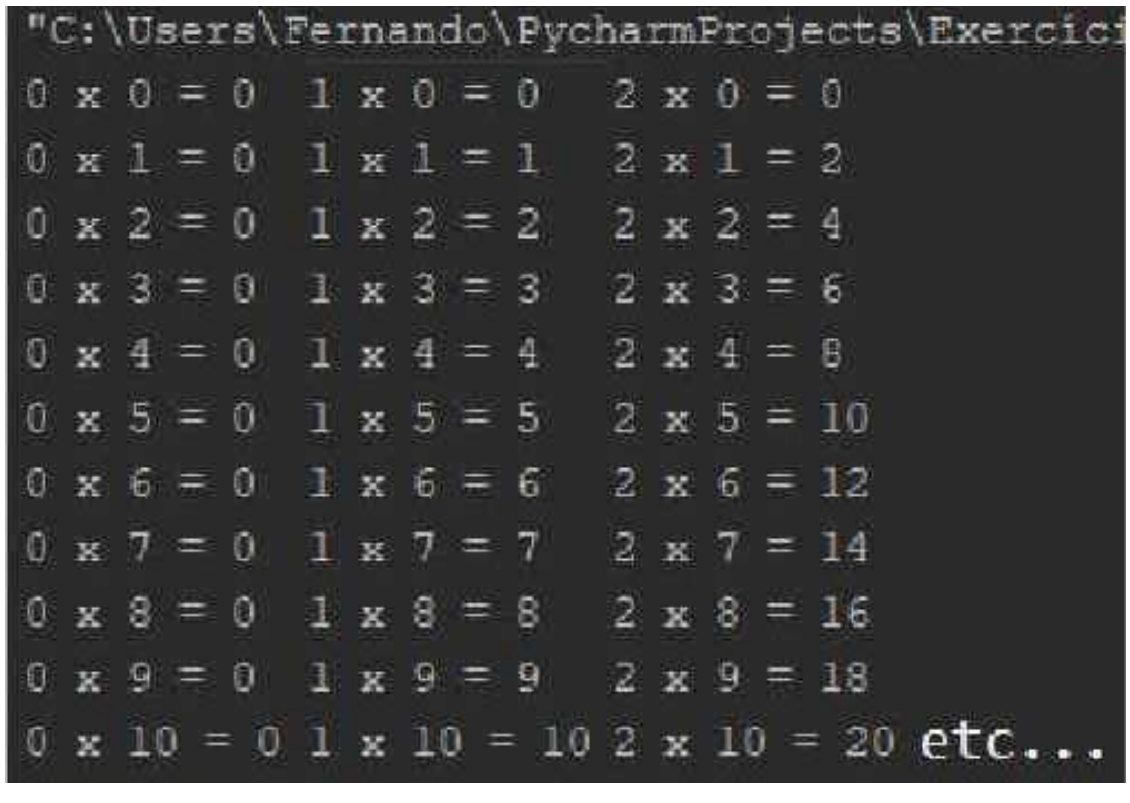
```
else:
    print('---Fim da lista!!!---')
```

O resultado será: **Pedro**
João
Leticia
---Fim da lista!!!---

Por fim, importante salientar que como for serve como laço de repetição ele suporta operações dentro de si, desde que essas operações requeiram repetidas instruções obviamente. Por exemplo:

```
for x in range(11):
    for y in range(11):
        print(f'{x} x {y} = {x * y}')
```

O retorno será toda a tabuada de **0 x 0** até **10 x 10** (que por convenção não irei colocar toda aqui obviamente...).



12 – Strings

Como já vimos algumas vezes ao longo da apostila, um objeto/variável é um espaço alocado na memória onde armazenaremos um tipo de dado ou informação para que o interpretador trabalhe com esses dados na execução do programa.

Uma **string** é o tipo de dado que usamos quando queremos trabalhar com qualquer tipo de texto, ou conjunto ordenado de caracretes alfanuméricos em geral. Quando atribuimos um conjunto de caracteres, representando uma palavra/texto, devemos obrigatoriamente seguir a sintaxe correta para que o interpretador leia os dados como tal.

A sintaxe para qualquer tipo de texto é basicamente colocar o conteúdo desse objeto entre aspas ' ', uma vez atribuido dados do tipo **string** para uma variavel, por exemplo **nome** = **'Maria'**, devemos lembrar de o referenciar como tal. Por exemplo em uma máscara lembrar de que o tipo de dado é **%s**.

Trabalhando com strings

No Python 3, se condicionou usar por padrão ' ' aspas simples para que se determine que aquele conteúdo é uma **string**, porém em nossa língua existem expressões que usam ' como apóstrofe, isso gera um erro de sintaxe ao interpretador. Por exemplo:

```
'marca d'água'
```

O retorno será um erro de sintaxe porque o interpretador quando abre aspas ele espera que feche aspas apenas uma vez, nessa expressão existem 3 aspas, o interpretador fica esperando que você "feche" aspas novamente, como isso não ocorre ele gera um erro.

O legal é que é muito fácil contornar uma situação dessas, uma vez que python suporta, com a mesma função, " "aspas duplas, usando o mesmo exemplo anterior, se você escrever "marca d'água" ele irá ler perfeitamente todos caracteres (incluindo o apóstrofe) como string. O mesmo ocorre se invertermos a ordem de uso das aspas. Por exemplo:

```
frase1 = 'Era um dia "muuuito" frio'

print(frase1)
```

O retorno será: **Era um dia "muuuito" frio**

Vimos anteriormente, na seção de comentários, que uma forma de comentar o código, quando precisamos fazer um comentário de múltiplas linhas, era as colocando entre """ aspas triplas (pode ser ''' aspas simples ou """ aspas duplas). Um texto entre aspas triplas é interpretado pelo interpretador como um comentário, ou seja, o seu conteúdo será por padrão ignorado, mas se atribuirmos esse texto de várias linhas a uma variável, ele passa a ser um objeto comum, do tipo **string**.

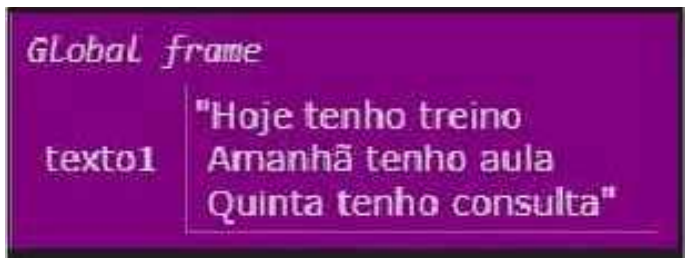
```
"""Hoje tenho treino
   Amanhã tenho aula
   Quinta tenho consulta"""
```

*Esse texto nessa forma é apenas um comentário.

```
texto1 = """Hoje tenho treino
            Amanhã tenho aula
            Quinta tenho consulta"""

print(texto1)
```

*Agora esse texto é legível ao interpretador, inclusive você pode printar ele ou usar da forma como quiser, uma vez que agora ele é uma **string**.



Formatando uma string

Quando estamos trabalhando com uma **string** também é bastante comum que por algum motivo precisamos formatá-la de alguma forma, e isso também é facilmente realizado desde que dados os comandos corretos.

Convertendo uma string para minúsculo

```
frase1 = 'A linguagem Python é muito fácil de aprender.'

print(frase1.lower())
```

Repare que na segunda linha o comando **print( )** recebe como parâmetro o conteúdo da variável **frase1** acrescido do comando **.lower( )**, convertendo a exibição dessa **string** para todos caracteres minúsculos.

O retorno será: **a linguagem python é muito fácil de aprender.**

Convertendo uma string para maiúsculo

Da mesma forma o comando **.upper( )** fará o oposto de **.lower( )**, convertendo tudo para maiúsculo. Ex:

```
frase1 = 'A linguagem Python é muito fácil de aprender.'

print(frase1.upper())
```

O retorno será: **A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.**

Lembrando que isso é uma formatação, ou seja, os valores da **string** não serão modificados, essa modificação não é permanente, ela é apenas a maneira com que você está pedindo para que o conteúdo da **string** seja mostrado.

Se você usar esses comandos em cima da variável em questão aí sim a mudança será permanente, por exemplo:

```
frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
frase1 = frase1.lower()

print(frase1)
```

O retorno será: **a linguagem python é muito fácil de aprender.**

Nesse caso você estará alterando permanentemente todos caracteres da **string** para minúsculo.

Buscando dados dentro de uma string

Você pode buscar um dado dentro do texto convertendo ele para facilitar achar esses dados, por exemplo um nome que você não sabe se lá dentro está escrito com a inicial maiúscula, todo em maiúsculo ou todo em minúsculo, para não ter que testar as 3 possibilidades, você pode usar comandos como:

```
frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
frase1 = frase1.lower()

print('py' in  frase1.lower())
```

O retorno será **True**. Primeiro toda **string** foi convertida para minúscula, por segundo foi procurado **'py'** que nesse caso consta dentro da **string**.

Desmembrando uma string

Outro comando interessante é o .**split( )**, ele irá desmembrar uma **string** em palavras separadas, para que você possa fazer a formatação ou o uso de apenas uma delas, por exemplo:

```
frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
frase1 = frase1.lower()

print(frase1.split())
```

O resultado será: **['a', 'linguagem', 'python', 'é', 'muito', 'fácil', 'de', 'aprender.']**

Alterando a cor de um texto

Supondo que você quer imprimir uma mensagem de erro que chame a atenção do usuário, você pode fazer isso associando também uma cor a este texto, por meio de um código de cores, é possível criar variáveis que irão determinar a cor de um texto, por exemplo:

```
ERRO = '\033[91m'   #código de cores "vermelho"
NORMAL  = '\033[0m'

print(ERRO, 'Mensagem de erro aqui' , NORMAL)
```

O Resultado será: **Mensagem de erro aqui.**

Uma vez que você declarou as cores como variáveis, você pode incorporar elas nos próprios parâmetros do código (note também que aqui como é uma variável criada para um caso especial, ela inclusive foi criada com nomenclatura toda maiúscula para eventualmente destaca-la de outras variáveis comuns no corpo do código).

```
print(ERRO + 'Mensagem de erro' + NORMAL)
```

O resultado será: **Mensagem de erro aqui.**

Apenas como referência, segue uma tabela com os principais códigos ANSI de cores tanto para a fonte quanto para o fundo.

Cor	Fonte	Fundo
Preto	\033[1;30m	\033[1;40m
Vermelho	\033[1;31m	\033[1;41m
Verde	\033[1;32m	\033[1;42m
Amarelo	\033[1;33m	\033[1;43m

Azul	\033[1;34m	\033[1;44m
Magenta	\033[1;35m	\033[1;45m
Cyan	\033[1;36m	\033[1;46m
Cinza Claro	\033[1;37m	\033[1;47m
Cinza Escuro	\033[1;90m	\033[1;100m
Vermelho Claro	\033[1;91m	\033[1;101m
Verde Claro	\033[1;92m	\033[1;102m
Amarelo Claro	\033[1;93m	\033[1;103m
Azul Claro	\033[1;94m	\033[1;104m
Magenta Claro	\033[1;95m	\033[1;105m
Cyan Claro	\033[1;96m	\033[1;106m
Branco	\033[1;97m	\033[1;107m

Alterando a posição de exibição de um texto

Estamos acostumados a, enquanto trabalhamos dentro de um editor de texto, usar do artifício de botões de atalho que podem fazer com que um determinado texto fique centralizado ou alinhado a um dos lados da página, por exemplo. Internamente isto é feito pelos comandos **center( )** (centralizar), **ljust( )** (alinhar à esquerda) e **rjust( )** (alinhar à direita).

Exemplo de centralização:

```
frase1 = 'Bem Vindo ao Meu Programa!!!'

print(frase1.center(50))
```

Repare que a função **print** aqui tem como parâmetro a variável **frase1** acrescida do comando **center(50)**, ou seja, centralizar dentro do intervalo de 50 caracteres. (A **string** inteira terá 50 caracteres, como **frase1** é menor do que isto, os outros caracteres antes e depois dela serão substituídos por espaços.

O retorno será: ‘ **Bem Vindo ao Meu Programa!!!** ‘

Exemplo de alinhamento à direita:

```
frase1 = 'Bem Vindo ao Meu Programa!!!'

print(frase1.rjust(50))
```

O retorno será: ‘ **Bem Vindo ao Meu Programa!!!**’

Formatando a apresentação de números em uma string

Existirão situações onde, seja em uma string ou num contexto geral, quando pedirmos ao Python o resultado de uma operação numérica ou um valor pré estabelecido na matemática (como pi por exemplo) ele irá exibir este número com mais casas decimais do que o necessário. Ex:

```
from math import pi

print(f'O número pi é: {pi}')
```

Na primeira linha estamos importando o valor de **pi** da biblioteca externa **math**. Em seguida dentro da **string** estamos usando uma máscara que irá ser substituída pelo valor de **pi**, mas nesse caso, como padrão.
O retorno será: **O número pi é: 3.141592653589793**

Num outro cenário, vamos imaginar que temos uma variável com um valor int extenso mas só queremos exibir suas duas primeiras casas decimais, nesse caso, isto pode ser feito pelo comando .f e uma máscara simples. Ex:

```
num1 = 34.295927957329247

print('O valor da ação fechou em %.2f'%num1)
```

Repare que dentro da **string** existe uma máscara de substituição **%** seguida de **.2f**, estes 2f significam ao interpretador que nós queremos que sejam exibidas apenas duas casas decimais após a vírgula. Neste caso o retorno será: **O valor da ação fechou em 34.30**

Usando o mesmo exemplo mas substituindo **.2f** por **.5f**, o resultado será: **O valor da ação fechou em 34.29593** (foram exibidas 5 casas “após a vírgula”).

13 – Listas

Listas em Python são o equivalente a Arrays em outras linguagens de programação, mas calma que se você está começando do zero, e começando com Python, esse tipo de conceito é bastante simples de entender.

Listas são um dos tipos de dados aos quais iremos trabalhar com frequencia, uma lista será um objeto que permite guardar diversos dados dentro dele, de forma organizada e indexada. É como se voce pegasse várias variáveis de vários tipos e colocasse em um espaço só da memória do seu computador, dessa maneira, com a indexação correta, o interpretador consegue buscar e ler esses dados de forma muito mais rapida do que trabalhar com eles individualmente. Ex:

```
lista = [ ]
```

Podemos facilmente criar uma lista com já valores inseridos ou adicionalos manualmente conforme nossa necessidade, sempre respeitando a sintaxe do tipo de dado. Ex:

```
lista2 = [1, 5, 'Maria', 'João']
```

Aqui criamos uma lista já com 4 dados inseridos no seu índice, repare que os tipos de dados podem ser mesclados, temos ints e strings na composição dessa lista, sem problema algum. Podemos assim deduzir também que uma lista é um tipo de variável onde conseguimos colocar diversos tipos de dados sem causar conflito.

Adicionando dados manualmente

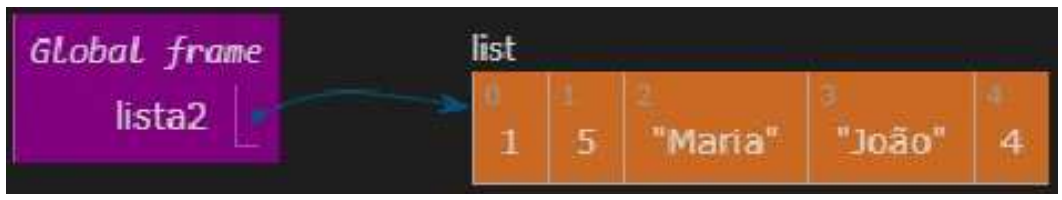
Se queremos adicionar manualmente um dado ou um valor a uma lista, este pode facilmente ser feito pelo comando **append( )**. Ex:

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)

print(lista2)
```

Irá adicionar o valor **4** na posição zero do indice da lista.

O retorno será: **[1, 5, 'Maria', 'João', 4]**



Lembrando que por enquanto, esses comandos são sequenciais, ou seja, executar o primeiro **.append** ira colocar um dado armazenado na posição 0 do índice, o segundo **.append** automaticamente ira guardar um dado na posição 1 do índice, e assim por diante...

Removendo dados manualmente

Da mesma forma, podemos executar o comando **.remove( )** para remover o conteúdo de algum índice da lista.

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)
lista2.remove('Maria')

print(lista2)
```

Irá remover o dado **Maria**, que nesse caso estava armazenado na posição **2** do indice.

O retorno será: **[1, 5, 'João', 4]**

Lembrando que no comando **.remove** você estará dizendo que conteúdo você quer excluir da lista, supondo que fosse uma lista de nomes **[‘Paulo’, ‘Ana’, ‘Maria’]** o comando **.remove(‘Maria’)** irá excluir o dado ‘**Maria’**, o proprio comando busca dentro da lista onde esse dado específico estava guardado e o remove, independentemente de sua posição.

Removendo dados via índice

Para deletarmos o conteúdo de um índice específico, basta executar o comando **del lista[nº do índice]**

```
lista2 = ['Ana', 'Carlos', 'João', 'Sonia']
del lista2[2]
```

```
print(lista2)
```

O retorno será: **['Ana', 'Carlos', 'Sonia']**

Repare que inicialmente temos uma lista **['Ana', 'Carlos', 'João', 'Sonia']** e executarmos o comando **del lista[2]** iremos deletar **'João'** pois ele está no índice **2** da lista. Nesse caso **'Sonia'** que era índice 3 passa a ser índice **2**, ele assume a casa anterior, pois não existe "espaço vazio" numa lista.

Verificando a posição de um elemento

Para verificarmos em que posição da lista está um determinado elemento, basta executarmos o comando **.index( )** Ex:

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)

print(lista2.index('Maria'))
```

O retorno será **2**, porque **'Maria'** está guardado no índice **2** da lista.

Repserentação visual:

Se você perguntar sobre um elemento que não está na lista o interpretador simplesmente retornará uma mensagem de erro dizendo que de fato, aquele elemento não está na nossa lista.

Verificando se um elemento consta na lista

Podemos também trabalhar com operadore aqui para consultar em nossa lista se um determinado elemento consta nela ou não. Por exemplo, executando o código **'João' in lista** devemos receber um retorno True ou False.

```
lista2 = [1, 5, 'Maria', 'João']
lista2.append(4)
```

```
print('João' in lista2)
```
O retorno será: **True**

Formatando dados de uma lista

Assim como usamos comandos para modificar/formatar uma **string** anteriormente, podemos aplicar os mesmos comandos a uma lista, porém é importante que fique bem claro que aqui as modificações ficam imediatamente alteradas (no caso da **string**, nossa formatação apenas alterava a forma como seria exibida, mas na variável a **string** permanecia integra). Ex:

```
lista1 = []
lista1.append('Paulo')
lista1.append('Maria')

print(lista1)
```
Inicialmente criamos uma lista vazia e por meio do método **append( )** inserimos nela duas **strings**. Por meio da função **print( )** podemos exibir em tela seu conteúdo:

O retorno será **[Paulo, Maria]**

Representação visual:


```
lista1.reverse()

print(lista1)
```
Se fizermos o comando **lista1.reverse( )** e em seguida dermos o comando **print(lista1)** o retorno será **[Maria, Paulo]**, e desta vez esta alteração será permanente.

Listas dentro de listas

Também é possível adicionar listas dentro de listas, parece confuso, mas na prática, ao inserir um novo dado dentro de um índice, se você usar a sintaxe de lista **[ ]** você estará adicionando, ali naquela posição do índice, uma nova lista. Ex:

```
lista = [1, 2, 4, 'Paulo']
```
Adicionando uma nova lista no índice 4 da lista atual ficaria:

```
lista = [1, 2, 4, 'Paulo']
lista = [1, 2, 4, 'Paulo', [2, 5, 'Ana']]

print(lista)
```

O retorno será **[1, 2, 4, 'Paulo', [2, 5, 'Ana']]**

Representação visual:

Tuplas

Uma Tupla trabalha assim como uma lista para o interpretador, mas a principal diferença entre elas é que lista é dinâmica (você pode alterá la à vontade) enquanto uma tupla é estática (elementos atribuídos são fixos) e haverão casos onde será interessante usar uma ou outra.

Um dos principais motivos para se usar uma tupla é o fato de poder incluir um elemento, o nome Paulo por exemplo, várias vezes em várias posições do índice, coisa que não é permitida em uma lista, a partir do momento que uma lista tem um dado ou valor atribuído, ele não pode se repetir.

Segunda diferença é que pela sintaxe uma lista é criada a partir de colchetes, uma tupla a partir de parênteses novamente.

```
minhatupla = tuple( )
```

Para que o interpretador não se confunda, achando que é um simples objeto com um parâmetro atribuído, pela sintaxe, mesmo que haja só um elemento na tupla, deve haver ao menos uma vírgula como separador. Ex:

```
minhatupla = tuple(1,)
```

Se você executar **type(minhatupla)** você verá **tuple** como retorno, o que está correto, se não houvesse a vírgula o interpretador iria retornar o valor int (já que 1 é um número inteiro).

Se você der um comando **dir(minhatupla)** você verá que basicamente temos **index** e **count** como comandos padrão que podemos executar, ou seja, ver seus dados e contá-los também.

Você pode acessar o índice exatmente como fazia com sua lista. Ex:

```
minhatupla = tuple(1,)
minhatupla[0]
```
O retorno será: **1**

Outro ponto a ser comentado é que, se sua tupla possuir apenas um elemento dentro de si, é necessário indicar que ela é uma tupla ou dentro de si colocar uma vírgula como separador mesmo que não haja um segundo elemento. Já quando temos 2 ou mais elementos dentro da tupla, não é mais necessário indicar que ela é uma tupla, o interpretador identificará automaticamente. Ex:

```
minhatupla = tuple('Ana')
minhatupla2 = ('Ana',)
minhatupla3 = ('Ana', 'Maria')
```
Em **minhatupla** existe a atribuição de tipo **tuple( ),** em **minhatupla2** existe a vírgula como separador mesmo não havendo um segundo elemento, em **minhatupla3** já não é mais necessário isto para que o identifique o tipo de dado como tupla.

Como uma tupla tem valores já predefinidos e imutáveis, é comum haver mais do mesmo elemento em diferentes posições do indice. Ex:

```
tuplacores = ('azul', 'branco', 'azul', 'vermelho', 'preto',
'azul', 'amarelo')
```
Executando o código **tuplacores.count('azul')**

```
tuplacores = ('azul', 'branco', 'azul', 'vermelho', 'preto',
'azul', 'amarelo')

print(tuplacores.count('azul'))
```
O retorno será **3** porque existem dentro da tupla '**azul**' 3 vezes.

Representação visual:

Pilhas

Pilhas nada mais são do que listas em que a inserção e a remoção de elementos acontecem na mesma extremidade. Para abstrairmos e entendermos de forma mais fácil, imagine uma pilha de papéis, se você colocar mais um papel na pilha, será em cima dos outros, da mesma forma que o primeiro papel a ser removido da pilha será o do topo.

Adicionando um elemento ao topo de pilha

Para adicionarmos um elemento ao topo da pilha podemos usar o nosso já conhecido **.append( )** recebendo o novo elemento como parâmetro**.**

```
pilha = [10, 20, 30]
pilha.append(50)

print(pilha)
```

O retorno será: **[10, 20, 30, 50]**

Removendo um elemento do topo da pilha

Para removermos um elemento do topo de uma pilha podemos usar o comando **.pop( )**, e neste caso como está subentendido que será removido o último elemento da pilha (o do topo) não é necessário declarar o mesmo como parâmetro**.**

```
pilha = [10, 20, 30]
pilha.pop()

print(pilha)
```

O retorno será: **[10, 20]**

Consultando o tamanho da pilha

Da mesma forma como consultamos o tamanho de uma lista, podemos consultar o tamanho de uma pilha, para termos noção de quantos elementos ela possui e de qual é o topo da pilha. Ex:

```
pilha = [10, 20, 30]
pilha.append(50)

print(pilha)
print(len(pilha))
```

O retorno será: **[10, 20, 30, 50]**
4

Representação visual:

Na primeira linha, referente ao primeiro **print( )**, nos mostra os elementos da pilha (já com a adição do "**50**" em seu topo. Por fim na segunda linha, referente ao segundo **print( )** temos o valor 4, dizendo que esta pilha tem 4 elementos;

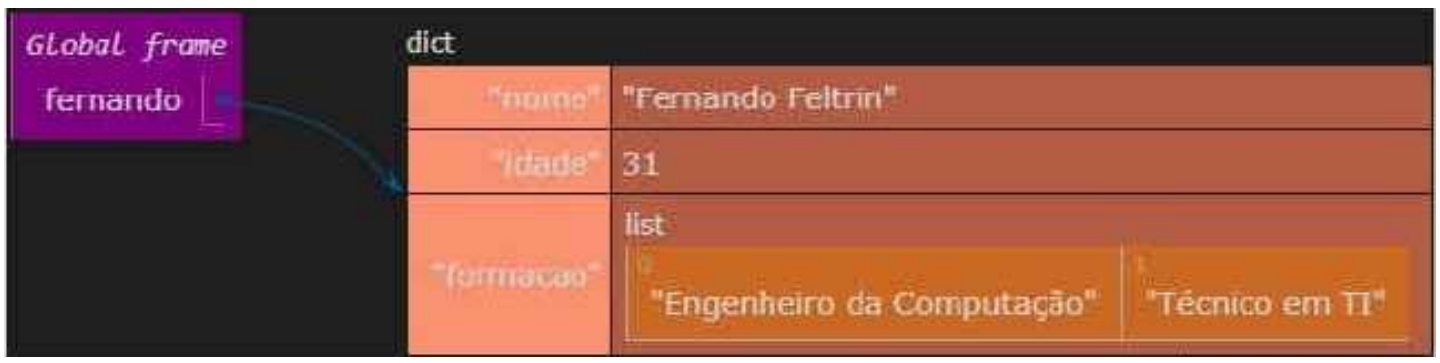
14 – Dicionários

Enquanto listas e tuplas são estruturas indexadas, um dicionário, além da sintaxe também diferente, se resume em **chave:valor**. Assim como em um dicionário normal você tem uma palavra e seu significado, aqui a estrutura lógica será essa.

A sintaxe de um dicionário é definida por chaves **{ }**, deve seguir a ordem **chave:valor** e usar vírgula como separador, para não gerar um erro de sintaxe. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31, 'formacao':
['Engenheiro da Computação', 'Técnico em TI']}
```

Repare que inicialmente o dicionário foi atribuido a um objeto, depois que seguindo a sintaxe foram criados 3 campos (nome, idade e formação) e dentro de formação ainda foi criado uma lista, onde foram adicionados dois elementos ao índice.



Uma prática comum tanto em listas, quanto em dicionários, é usar de uma tabulação que torne visualmente o código mais organizado, sempre que estivermos trabalhando com estes tipos de dados teremos uma vírgula como separador dos elementos, e é perfeitamente normal após uma vírgula fazer uma quebra de linha para que o código fique mais legível. Esta prática não afeta em nada a leitura léxica do interpretador nem a performance de operação do código.

```
fernando = {'nome': 'Fernando Feltrin',
            'idade': 31,
            'formacao': ['Engenheiro da Computação',
                         'Técnico em TI']}
```

Executando um **type(fernando)** o retorno será **dict**, porque o interpretador, tudo o que estiver dentro de **{ }** chaves ele interpretará como chaves e valores de um dicionário.

Assim como existem listas dentro de listas, você pode criar listas dentro de dicionários e até mesmo dicionários dentro de dicionários sem problema algum.

Você pode usar qualquer tipo de dado como chave e valor, o que não pode acontecer é esquecer de declarar um ou outro pois irá gerar erro de sintaxe.

Da mesma forma como fazíamos com listas, é interessante saber o tamanho de um dicionário, e assim como em listas, o comando **len( )** agora nos retornará a quantidade de **chaves:valores** inclusos no dicionário. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,
            'formacao': ['Engenheiro da Computação', 'Téc em
TI']}

print(len(fernando))
```

O retorno será **3**.

Consultando chaves/valores de um dicionário

Você pode consultar o valor de dentro de um dicionário pelo comando **.get( ).** Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,
            'formacao': ['Engenheiro da Computação', 'Técnico em
TI']}

print(fernando.get('idade'))
```

O retorno será **31.**

Consultando as chaves de um dicionário

É possível consultar quantas e quais são as chaves que estão inclusas dentro de um dicionário pelo comando .**keys( ).** Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,
            'formacao': ['Engenheiro da Computação', 'Técnico em
TI']}

print(fernando.keys())
```

O retorno será: **dict_keys(['nome', 'idade', 'formacao'])**

Consultando os valores de um dicionário

Assim como é possível fazer a leitura somente dos valores, pelo comando .**values( )**. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,
            'formacao': ['Engenheiro da Computação', 'Técnico em
```

```
TI']}

print(fernando.values())
```

O retorno será: **dict_values(['Fernando Feltrin', 31, ['Engenheiro da Computação', 'Técnico em TI']])**

Mostrando todas chaves e valores de um dicionário

Por fim o comando **.items( )** retornará todas as chaves e valores para sua consulta. Ex:

```
fernando = {'nome': 'Fernando Feltrin', 'idade': 31,
            'formacao': ['Engenheiro da Computação', 'Técnico em
TI']}

print(fernando.items())
```

O retorno será: **dict_items([('nome': 'Fernando Feltrin', 'idade': 31, 'formacao': ['Engenheiro da Computação', 'Técnico em TI'])])**

Manipulando dados de um dicionário

Quanto temos um dicionário já com valores definidos, pre-programados, mas queremos alterá-los, digamos, atualizar os dados dentro de nosso dicionario, podemos fazer isso manualmente atraves de alguns simples comandos.

Digamos que tinhamos um dicionario inicial:

```
pessoa = {'nome': 'Alberto Feltrin',
          'idade': '42',
          'formação': ['Tec. em Radiologia'],
          'nacionalidade': 'brasileiro'}

print(pessoa)
```

O retorno será: **{'nome': 'Alberto Feltrin', 'idade': '42', 'formação': ['Tec. em Radiologia'], 'nacionalidade': 'brasileiro'}**

Representação visual:


```
pessoa = {'nome': 'Alberto Feltrin',
          'idade': '42',
          'formação': ['Tec. em Radiologia'],
          'nacionalidade': 'brasileiro'}

pessoa['idade'] = 44

print(pessoa)
```

Executando o comando **pessoa['idade'] = 44** estaremos atualizando o valor '**idade**' para **44** no nosso dicionário, consultando o dicionário novamente você verá que a idade foi atualizada.

Executando novamente **print(pessoa)** o retorno será: **{'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia', 'nacionalidade']: 'brasileiro'}**

Adicionando novos dados a um dicionário

Além de substituir um valor por um mais atual, também é possível adicionar manualmente mais dados ao dicionário, assim como fizemos anteriormente com nossas listas, através do comando .**append( )**. Ex:

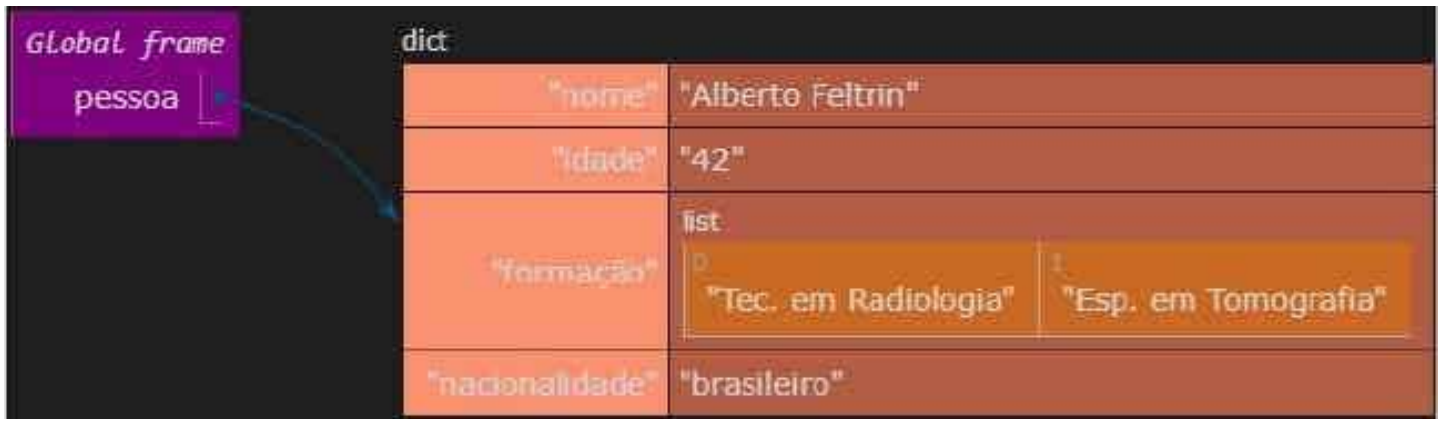
```
pessoa = {'nome': 'Alberto Feltrin',
          'idade': '42',
          'formação': ['Tec. em Radiologia'],
          'nacionalidade': 'brasileiro'}

pessoa['formação'].append('Esp. em Tomografia')

print(pessoa)
```

O retorno será: **{'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia', 'Esp em Tomografia'], 'nacionalidade': 'brasileiro'}**

Representação visual:



*Importante salientar que para que você possa adicionar mais valores a uma chave, ela deve respeitar a sintaxe de uma lista.

15 – Conjuntos numéricos

Se você lembrar das aulas do ensino médio certamente lembrará que em alguma etapa lhe foi ensinado sobre conjuntos numéricos, que nada mais era do que uma foram de categorizarmos os números qanto suas características (reais, inteiros, etc...). Na programação isto se repete de forma parecida, uma vez que quando queremos trabalhar com conjuntos numéricos normalmente estamos organizando nú meros.

Um conjunto é ainda outra possibilidade de armazenamento de dados que temos, de forma que ele parece uma lista com sintaxe de dicionário, mas não indexável e que não aceita valores repetidos, confuso não? A questão é que como aqui podemos ter valores repetidos, assim como conjuntos numéricos que usávamos no ensino médio, aqui podemos unir dois conjuntos, fazer a intersecção entre eles, etc... Vamos aos exemplos:

```
a = {1, 2, 3}
```

Se executarmos o comando **type(a)** o retorno será **set**. O interpretador consegue ver na sintaxe que não é nem uma lista, nem uma tupla e nem um dicionário, mas apenas um conjunto de dados alinhados.

Trabalhando com mais de um conjunto, podemos fazer operações entre eles. Por exemplo:

União de conjuntos

A união de dois conjuntos é feita através do operador **.union( )**, de forma que a operação irá na verdade gerar um terceiro conjunto onde constam os valores dos dois anteriores. Ex:

```
c1 = {1, 2}
c2 = {2, 3}
c1.union(c2)
#c1 união com c2, matematicamente juntar 2 conjuntos.
```

O retorno será: **{1, 2, 3}**

Repare que a união dos dois conjuntos foi feita de forma a pegar os valores que não eram comuns aos dois conjuntos e incluir, e os valores que eram comuns aos dois conjuntos simplesmente manter, sem realizar nenhuma operação.