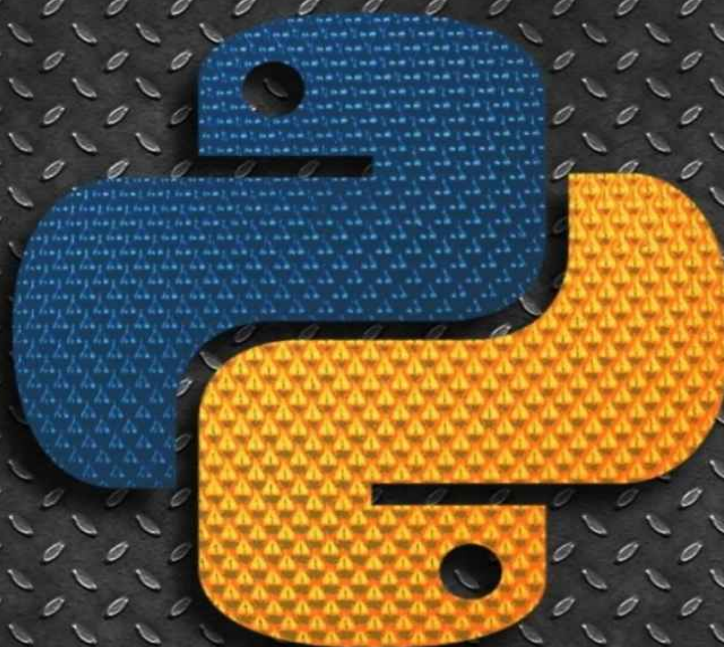




Python do **ZERO** à Programação Orientada a Objetos

Fernando Feltrin



Python do ZERO

à Programação Orientada a Objetos

Fernando Feltrin


Uniorg

Python do ZERO à Programação Orientada a Objetos

Fernando Feltrin

Editora Uniorg

AVISOS

Este livro conta com mecanismo anti pirataria Amazon Kindle Protect DRM. Cada cópia possui um identificador próprio rastreável, a distribuição ilegal deste conteúdo resultará nas medidas legais cabíveis.

É permitido o uso de trechos do conteúdo para uso como fonte desde que dados os devidos créditos ao autor.

SOBRE O AUTOR



Fernando Feltrin é Engenheiro da Computação com especializações na área de ciência de dados e inteligência artificial, Professor licenciado para docência de nível técnico e superior, Autor de mais de 10 livros sobre programação de computadores e responsável por desenvolvimento e implementação de ferramentas voltadas a modelos de redes neurais artificiais aplicadas à radiologia (diagnóstico por imagem).

ÍNDICE

[AVISOS](#)

[SOBRE O AUTOR](#)

[ÍNDICE](#)

[Por quê programar? E por quê em Python?](#)

[Metodologia](#)

[INTRODUÇÃO](#)

[Por quê Python?](#)

[Um pouco de história](#)

[Guido Van Rossum](#)

[A filosofia do Python](#)

[Empresas que usam Python](#)

[O futuro da linguagem](#)

[Python será o limite?](#)

[AMBIENTE DE PROGRAMAÇÃO](#)

[Linguagens de alto, baixo nível e de máquina](#)

[Ambientes de desenvolvimento integrado](#)

[Principais IDEs](#)

[LÓGICA DE PROGRAMAÇÃO](#)

[Algoritmos](#)

[Sintaxe em Python](#)

[Palavras reservadas](#)

[Análise léxica](#)

[Indentação](#)

ESTRUTURA BÁSICA DE UM PROGRAMA

TIPOS DE DADOS

COMENTÁRIOS

VARIÁVEIS / OBJETOS

Declarando uma variável

Declarando múltiplas variáveis

Declarando múltiplas variáveis (de mesmo tipo)

FUNÇÕES BÁSICAS

Função print(.)

Função input(.)

Explorando a função print(.)

Interação entre variáveis

Conversão de tipos de dados

OPERADORES

Operadores de Atribuição

Atribuições especiais

Operadores aritméticos

Soma

Subtração

Multiplicação

Divisão

Exponenciação

Operadores Lógicos

Tabela verdade

Tabela Verdade OR (OU)

[Tabela Verdade XOR \(OU Exclusivo/um ou outro\)](#)

[Tabela de Operador de Negação \(unário\)](#)

[Bit-a-bit](#)

[Operadores de membro](#)

[Operadores relacionais](#)

[Operadores usando variáveis](#)

[Operadores usando condicionais](#)

[Operadores de identidade](#)

[ESTRUTURAS CONDICIONAIS](#)

[Ifs, elifs e elses](#)

[And e Or dentro de condicionais](#)

[Condicionais dentro de condicionais](#)

[Simulando switch/case](#)

[ESTRUTURAS DE REPETIÇÃO](#)

[While](#)

[For](#)

[STRINGS](#)

[Trabalhando com strings](#)

[Formatando uma string](#)

[Convertendo uma string para minúsculo](#)

[Convertendo uma string para maiúsculo](#)

[Buscando dados dentro de uma string](#)

[Desmembrando uma string](#)

[Alterando a cor de um texto](#)

[Alterando a posição de exibição de um texto](#)

Formatando a apresentação de números em uma string

LISTAS

Adicionando dados manualmente

Removendo dados manualmente

Removendo dados via índice

Verificando a posição de um elemento

Verificando se um elemento consta na lista

Formatando dados de uma lista

Listas dentro de listas

Trabalhando com Tuplas

Trabalhando com Pilhas

Adicionando um elemento ao topo de pilha

Removendo um elemento do topo da pilha

Consultando o tamanho da pilha

DICIONÁRIOS

Consultando chaves/valores de um dicionário

Consultando as chaves de um dicionário

Consultando os valores de um dicionário

Mostrando todas chaves e valores de um dicionário

Manipulando dados de um dicionário

Adicionando novos dados a um dicionário

CONJUNTOS NUMÉRICOS

União de conjuntos

Interseção de conjuntos

Verificando se um conjunto pertence ao outro

[Diferença entre conjuntos](#)

[INTERPOLAÇÃO](#)

[Avançando com interpolações](#)

[FUNÇÕES](#)

[Funções predefinidas](#)

[Funções personalizadas](#)

[Função simples, sem parâmetros](#)

[Função composta, com parâmetros](#)

[Função composta, com *args e **kwargs](#)

[dir\(.\) e help\(.\)](#)

[BUILTINS](#)

[Importando bibliotecas](#)

[MÓDULOS E PACOTES](#)

[Modularização](#)

[Importando de módulos](#)

[PROGRAMAÇÃO ORIENTADA A OBJETOS](#)

[Classes](#)

[Definindo uma classe](#)

[Alterando dados/valores de uma instância](#)

[Aplicando recursividade](#)

[Herança](#)

[Polimorfismo](#)

[Encapsulamento](#)

[TRACEBACKS / EXCEÇÕES](#)

[Comandos try, except e finally.](#)

CONSIDERAÇÕES FINAIS

LIVROS

CURSO

REDES SOCIAIS

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 vasta gama 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 de nossa “informática”, um dos nichos diferenciais é a área da programação, tanto pela sua complexidade quanto por sua enorme 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, seja ela em linguagens generalistas ou especialistas. No âmbito acadêmico existem diversos cursos, que vão de análise e desenvolvimento de sistemas até engenharia da computação, e observando a 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 a C ou uma de suas vertentes, isso se dá pelo fato de C ser uma linguagem ainda hoje bastante popular (se não a mais popular em aplicações), também podendo servir de base para muitas 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 de aplicações específicas. 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, visão computacional, além é claro da criação de aplicações web, aplicativos mobile, softwares, jogos e sistemas comerciais, mais do que nunca foi preciso profissionais da área que possuam ao menos noções de 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 graças a suas ferramentas e capacidade de se adaptar para qualquer propósito.

Python pode ser a sua linguagem de programação definitiva, ou abrir muitas portas para aprender outras mais, já que nesta área 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, linha por linha, 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!!!

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, pois haverá uma sintaxe e uma sequência lógica de argumentos a se respeitar a fim de que seus comandos façam sentido e sejam interpretados pelo núcleo da linguagem, assim como em uma linguagem natural e escrita, organizamos sílabas e fonemas de modo a reproduzir ideias a serem compreendidas por outra pessoa.

Com certeza quando você iniciou seu processo de alfabetização, começou a aprender letras, sílabas, palavras, regras gramaticais, etc... tudo para usar da linguagem uma ferramenta de comunicação e interação com o mundo ao redor. Ao buscar aprender outro idioma, inglês por exemplo, também começou pelo básico de sua gramática, o verbo to be, e posteriormente foi incrementando com novos conceitos e vocabulário, até ter certo domínio sobre tal língua.

Em uma linguagem de programação não é muito diferente, há uma sintaxe, equivalente a gramática, 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 para executar uma determinada ação ou gerar um 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 nossos 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 de escrita, e seu potencial de fazer mais com menos no que diz respeito a suas ferramentas nativas. Não desmerecendo outras linguagens de programação, mas é fato que Python foi desenvolvida para ser descomplicada e ao mesmo tempo suprir necessidades de resolução de problemas computacionais com facilidade e eficiência.

Aqui cabe facilmente a pergunta:

- E por quê fazer as coisas da forma mais difícil se existem ferramentas para torná-las mais fáceis?

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

“- A vida é curta demais para programar em outra linguagem de programação, 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 por fim em Python.

Hello World!!! em C

```
1  #include <stdio.h>
2
3  int main(void) {
4      printf("Hello World\n");
5      return 0;
6  }
7
```

Hello World!!! em JAVA

```
1  class Main {  
2      public static void main(String[] args) {  
3          System.out.println("Hello world!");  
4      }  
5  }  
6
```

Hello World!!! em Python

```
1  print('Hello World!')  
2
```

Em todos os exemplos acima o retorno será uma mensagem exibida em tela para o usuário dizendo: Hello World!!! (*Garanto que agora a piada sobre outras linguagens 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 você conhecerá este padrão irá se repetir. Em Python será muito mais fácil, mais rápido, e com menos linhas de código para se atingir o objetivo final.

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.

Python preza por uma simplicidade sintática, posteriormente você aprenderá sobre a sintaxe Python, mas por hora apenas imagine que em programação (outras linguagens) muitas vezes temos uma grande ideia e ao codificá-la acabamos nos frustrando porque o código simplesmente não funciona, e em boa parte das vezes é por conta de algum pequeno erro de escrita (um ponto ou

vírgula que ficou sobrando ou faltando no código), tipo de situação que não ocorre em Python.

Python por ser uma linguagem interpretada deveria sofrer ainda mais com esses problemas, mas o que ocorre é justamente o contrário, o interpretador foca nos comandos e seus parâmetros, os pequenos erros de sintaxe não farão com que o código não funcione, apenas serão emitidos alertas para que em tempo real se corrija o(s) erro(s).

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 processando volumes enormes de dados.

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

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 da linguagem C, de modo que 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 considerável comunidade de entusiastas dedicada a aprimorá-la.

Somente em 1994 foi lançada sua versão 1.0, ou seja, sua primeira versão totalmente funcional, e não mais de testes, e de lá para cá houveram gigantescas melhorias na linguagem em si, seja em estrutura quanto em bibliotecas e plugins criadas pela comunidade para implementar novos recursos para a mesma tornando-a cada vez 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 trás 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 graças a seus feitos e contribuições para a comunidade.

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 pelo seu núcleo de desenvolvedores quanto pela comunidade, e ainda hoje Guido continua supervisionando o desenvolvimento da linguagem Python, tomando certas 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 ambiguidade, 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 frequentemente melhor que **agora mesmo**.
17. Se a implementação é difícil para explicar, isto é uma ideia ruim.
18. Se a implementação é fácil para explicar, pode ser uma ideia boa.
19. Namespaces são uma grande ideia -- façamos mais desses!

Essa filosofia é o que fez com que se agregasse uma comunidade enorme disposta a investir seu tempo e esforços na linguagem Python. Em suma, programar em Python deve ser simples, de fácil aprendizado, com código de fácil leitura, enxuto, mas inteligível, eficiente e 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 dividi-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 ou a linguagem R, específica para ciência de dados. Já linguagens generalistas são aquelas que tem sua aplicabilidade em todo e qualquer propósito, e nem por isso ser inferior às específicas, apenas possuindo recursos e ferramentas facilmente adaptáveis a qualquer implementação.

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, programação web, mobile, data science, machine learning, blockchain, visão computacional, etc... coisa que outras linguagens de forma nativa não são suficientes ou ao menos práticas para o programador, necessitando uma série de gambiarras para realizar sua codificação, tornando o processo mais difícil ou menos eficiente.

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 incrível animações e filmes compostos de 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 e suas bibliotecas.

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, assim como computação distribuída em sistemas como blockchain, 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, muito pelo contrário, uma vez que todas tecnologias citadas acima podem ser facilmente implementadas via Python.

A versão 3 do Python é por si só bastante robusta e consegue de forma nativa trabalhar com tais tecnologias, haja visto que devido a enorme comunidade que desenvolve ou aprimora ferramentas para Python, podemos esperar que para o futuro certamente haverá novas versões da linguagem implementando novos recursos de forma nativa. Será muito difícil vermos surgir outra linguagem “do zero” ou que tenha usabilidade parecida com Python, uma vez que a mesma é bastante consolidada no mercado graças a sua robustez.

Se você olhar para trás no tempo, verá uma série de linguagens que foram descontinuadas ou que simplesmente foram perdendo seu espaço, mesmo seus desenvolvedores insistindo e injetando tempo e dinheiro em seu desenvolvimento, muitas delas não eram modernas o suficiente ou não acompanharam a rápida evolução deste nicho da computação. Do meu ponto de vista não consigo ver um cenário do futuro ao qual Python não consiga se adaptar ou até mesmo cenário onde Python não seja destaque entre as demais.

Entenda que 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, coisa que Python nos proporciona.

Num futuro próximo nosso diferencial como profissionais da área de tecnologia será ter conhecimento sobre linguagens de programação que se adaptem as necessidades do mercado. Garanto que você já sabe em qual delas estou apostando minhas fichas...

AMBIENTE DE PROGRAMAÇÃO

Na linguagem Python, e, não diferente das outras linguagens de programação, quando partimos do campo das ideias 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 (interface que interage com o usuário final).

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, basicamente 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 linguísticos e conceitos normalmente vindos do inglês e que o usuário pode pegar qualquer bloco de código de modo que 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 próprio programados. Em outras palavras, enquanto em alto nível temos códigos que abstraem ideias de forma legível, em baixo nível o código em execução terá instruções para o processamento de dados alocados em memória e em threads no próprio processador.

Quando estamos programando em Python sempre estaremos num ambiente de linguagem de alto nível, onde usaremos expressões em inglês e uma sintaxe fácil para por fim dar nossas instruções ao computador de que dados deverão ser processados ou de que ações deverão serem executadas a partir de nosso código. Esta linguagem posteriormente será convertida para camadas de linguagem de baixo nível que por fim se tornarão sequências de instruções para registradores, portas lógicas ou até mesmo chaveamentos de circuitos dependendo a usabilidade.

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-o 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.

Já na era da computação, 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 de um 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 em nosso sistema operacional temos a frontend do sistema, ou seja, a camada 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 tendo que emular estruturas de acesso ao hardware, assim como se programava separado de onde se compilava, separado de onde se debugava e por fim separado de onde se rodava o programa finalmente vendo o ver usando dos recursos do sistema operacional e executando suas devidas funcionalidades. Apenas entenda que as IDEs unificam todas camadas necessárias emulando tudo o que é necessário para que possamos nos concentrar em apenas escrever nossos códigos e executá-los ao final desse processo.

Entenda que é possível programar em qualquer editor de texto, como o próprio bloco de notas do sistema operacional ou em um terminal, porém o uso de IDEs se dá pela facilidade de possuir todo um espectro de ferramentas dedicadas de forma integrada em um mesmo ambiente.

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 editor de texto, 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 para fins de estudo, sendo bastante intuitiva e fácil de aprender a usar seus recursos, por fim, ela oferece um ambiente com suporte em tempo real ao uso de console/terminal próprio, assim como capacidade de gerir ambientes virtualizados isolados para cada projeto, sendo assim possível a qualquer momento executar testes nos seus blocos de código independentemente se os mesmos usam apenas recursos nativos da plataforma ou se usam de bibliotecas instaladas exclusivamente para o projeto.

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

Anaconda – Já a suíte Anaconda, também gratuita, conta com uma série de ferramentas que se destacam por suas aplicabilidades. Python, como já mencionado diversas vezes, é uma linguagem de programação 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 de, é claro, do próprio ambiente integrado de programação

padrão, que neste caso é o VisualStudioCode, entre outras ferramentas todas pré-configuradas para interação entre si.

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

Importante salientar também que, é perfeitamente normal e possível programar direto a partir de um terminal, independentemente do sistema operacional, ou em terminais oferecidos pelas próprias IDEs, a escolha de usar um ou outro é de particularidade do próprio programador.

Como dito anteriormente, existem diversas IDEs e ferramentas que facilitarão sua vida como programador, a verdade é que é interessante você dedicar um tempinho a testar algumas IDEs e ver qual você se adapta melhor tanto pela facilidade de uso quanto pelas ferramentas disponibilizadas de acordo com sua implementação.

Não existe uma IDE melhor que a outra, o que existem são ferramentas que se adaptam melhor as suas necessidades enquanto programador.

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 chamamos de linguagem os meios que criamos para conseguir passar comandos a serem interpretados e executados em um computador.

Quando falamos sobre um determinado assunto, automaticamente em nossa língua falada nativa geramos uma sentença que possui uma certa sintaxe gramatical 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 corretamente realizado.

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

Algoritmos

Todo estudante de computação no início de seu curso recebe uma boa carga horária de algoritmos. Algoritmos em suma nada mais são do que uma sequência de passos ordenados 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 e em que quantidade 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 funcionalidade, temos que passar as instruções passo a passo para que o interpretador consiga as executar e chegar ao fim de um determinado 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 neste primeiro momento, 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 toda a mistura já pronta de ingredientes, bastando apenas adicionar leite e colocar para assar (neste caso, não temos as entradas de forma explícita, 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 acondicionados em um invólucro todo o necessário para produzir o

bolo, porém não é você que fará o bolo (neste caso temos as entradas e nenhuma saída).

Em nossos programas haverá situações onde iremos criar scripts que serão executados (de forma explícita ou implícita para o usuário) de acordo com as entradas necessárias e as saídas esperadas para resolver algum tipo de problema computacional específico.

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 ideia, 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 ideia 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:

A screenshot of a Python interpreter window. The top bar has a play button icon. The main area shows two lines of code: `1 print('5' + '2')` and `2`. The bottom bar shows a prompt `>` followed by the number `52`.

```
1 print('5' + '2')
2
> 52
```

O resultado será 52, porque de acordo com a sintaxe do Python 3, tudo o que digitamos entre ‘aspas’ é uma string (um 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:



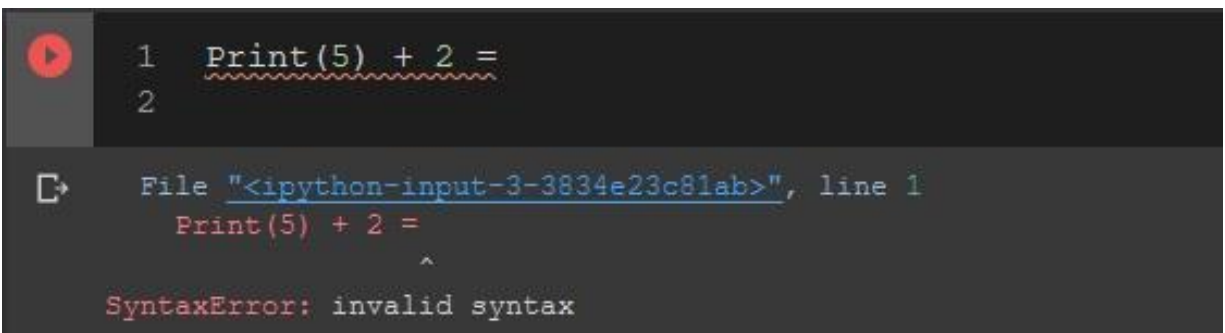
```
1 print(5 + 2)
2
```

7

The screenshot shows a Jupyter Notebook interface. The top part is a code editor with two lines of Python code: `1 print(5 + 2)` and `2`. The bottom part is the output area, which displays the number `7`. A play button icon is visible on the left side of the code editor.

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:



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

File "<ipython-input-3-3834e23c81ab>", line 1
Print(5) + 2 =
 ^
SyntaxError: invalid syntax

The screenshot shows a Jupyter Notebook interface. The top part is a code editor with two lines of Python code: `1 Print(5) + 2 =` and `2`. The bottom part is the output area, which displays an error message: `SyntaxError: invalid syntax`. A red play button icon is visible on the left side of the code editor.

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 espera para poder 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.

À medida que formos progredindo você irá automaticamente associar que determinadas “palavras” são comandos ou instruções internas a linguagem. Você precisa falar uma língua 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 ideia 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.

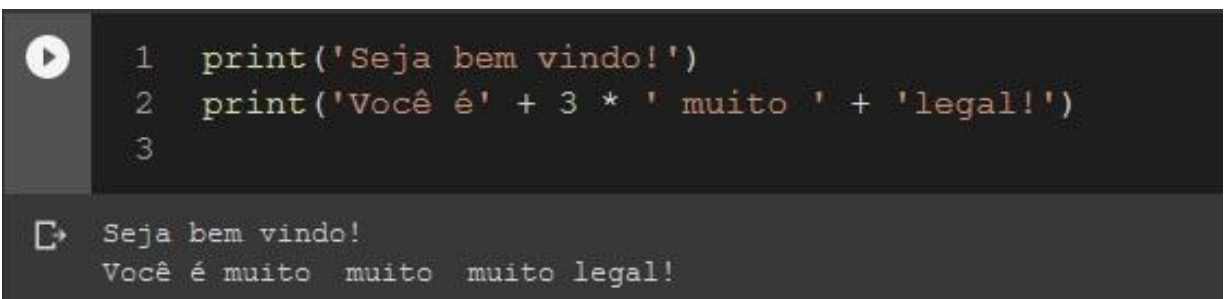
Raciocine que todo e qualquer código deve seguir uma sequência lógica, um passo-a-passo a ser seguido em uma ordem definida, basicamente quando estamos programando dividimos cada passo de nosso código em uma linha nova/diferente.

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 sentido 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:



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

Seja bem vindo!
Você é muito muito 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.

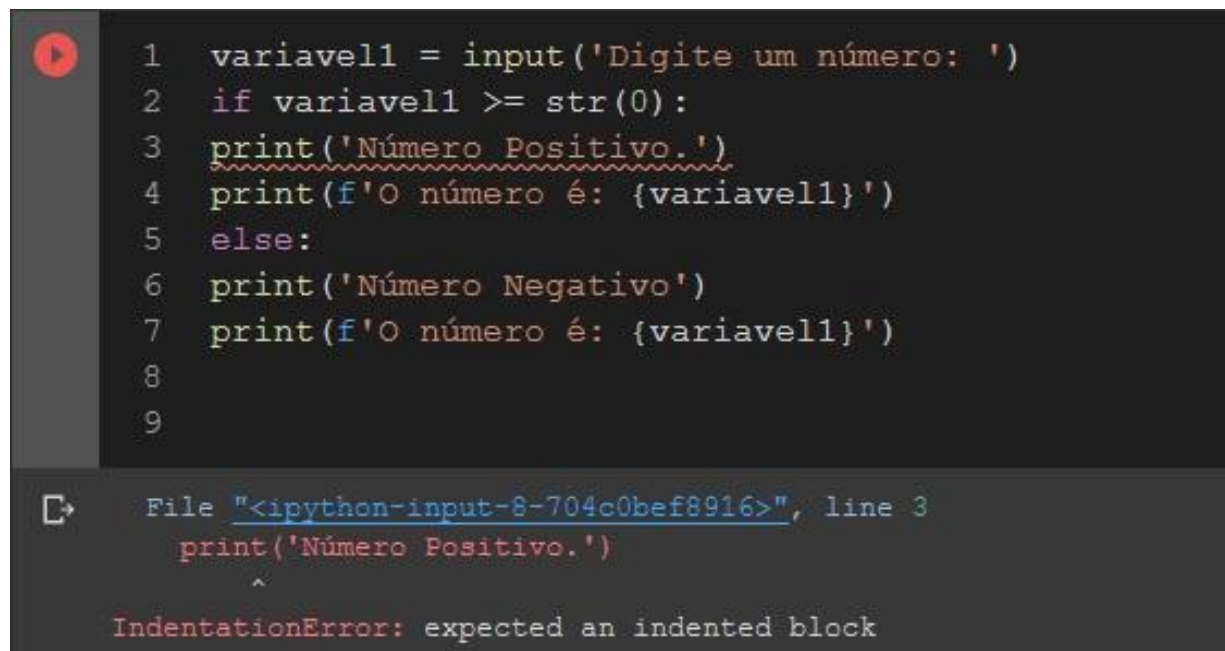
Indentação

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:

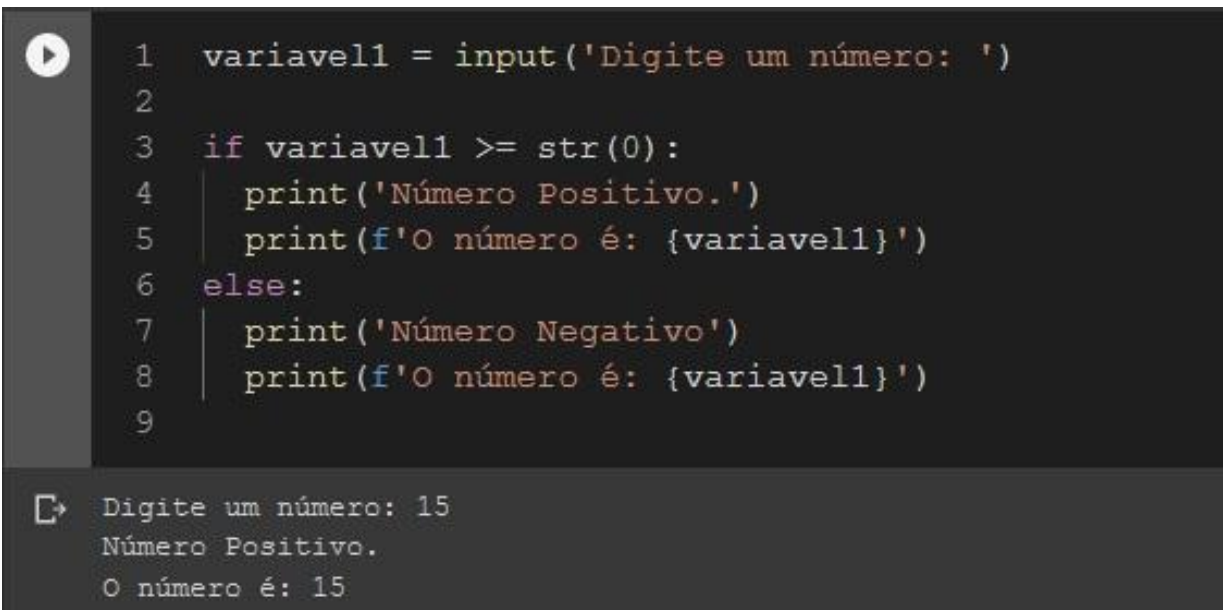


```
1  variavell = input('Digite um número: ')
2  if variavell >= str(0):
3  print('Número Positivo.')
4  print(f'O número é: {variavell}')
5  else:
6  print('Número Negativo')
7  print(f'O número é: {variavell}')
8
9
```

File "<ipython-input-8-704c0bef8916>", line 3
print('Número Positivo.')
^
IndentationError: expected an indented block

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:



```
1  variavel1 = input('Digite um número: ')
2
3  if variavel1 >= str(0):
4      print('Número Positivo.')
5      print(f'O número é: {variavel1}')
6  else:
7      print('Número Negativo')
8      print(f'O número é: {variavel1}')
9
```

↳ Digite um número: 15
Número Positivo.
O número é: 15

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.

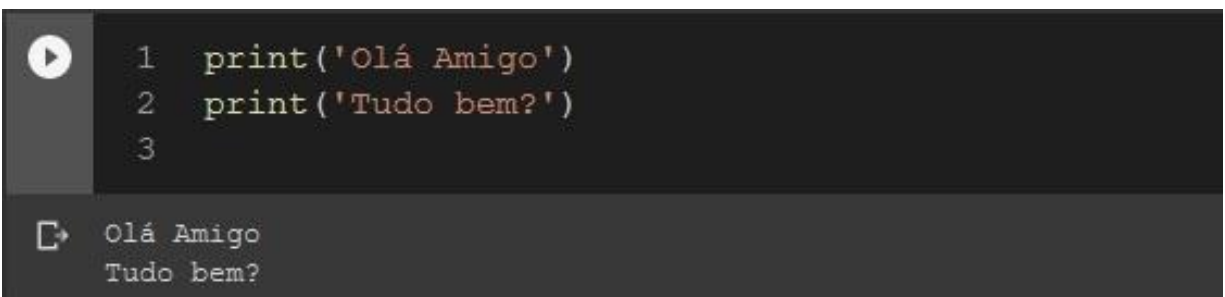
O mesmo ocorre por parte do interpretador, de acordo com a tabulação usada, ele consegue perceber que inicialmente existe uma variável `variavel1` declarada que pede que o usuário digite um número, também que existem duas estruturas condicionais `if` e `else`, e que dentro delas existem instruções de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

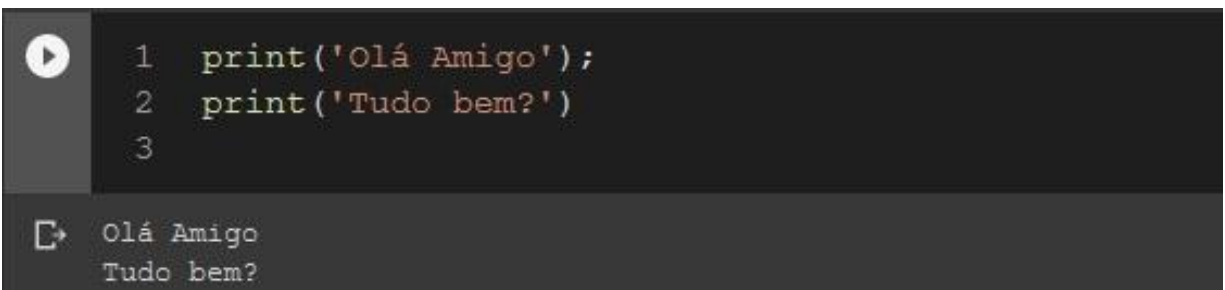


```
1 print('Olá Amigo')
2 print('Tudo bem?')
3
```

Olá Amigo
Tudo bem?

Detailed description: A screenshot of a Python code editor with a dark background. The code consists of two lines: `1 print('Olá Amigo')` and `2 print('Tudo bem?')`, followed by a blank line `3`. To the left of the code is a play button icon. Below the code editor, the output is displayed: `Olá Amigo` on the first line and `Tudo bem?` on the second line, preceded by a copy icon.

Código com vício de linguagem:

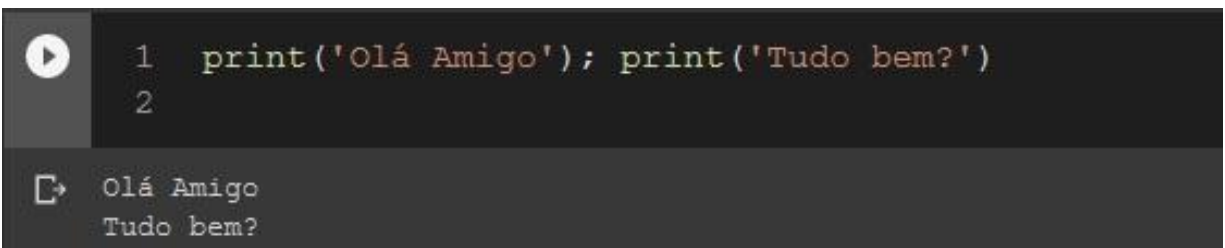


```
1 print('Olá Amigo');
2 print('Tudo bem?')
3
```

Olá Amigo
Tudo bem?

Detailed description: A screenshot of a Python code editor with a dark background. The code consists of two lines: `1 print('Olá Amigo');` and `2 print('Tudo bem?')`, followed by a blank line `3`. To the left of the code is a play button icon. Below the code editor, the output is displayed: `Olá Amigo` on the first line and `Tudo bem?` on the second line, preceded by a copy icon.

Código extrapolado:



```
1 print('Olá Amigo'); print('Tudo bem?')
2
```

Olá Amigo
Tudo bem?

Detailed description: A screenshot of a Python code editor with a dark background. The code consists of two lines: `1 print('Olá Amigo'); print('Tudo bem?')` and a blank line `2`. To the left of the code is a play button icon. Below the code editor, the output is displayed: `Olá Amigo` on the first line and `Tudo bem?` on the second line, preceded by a copy icon.

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?

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:

A screenshot of a Python IDE interface. The top part shows a code editor with a single line of Python code: `1 print('Olá Mundo!!!')`. To the left of the code is a play button icon. Below the code editor is a console window that displays the output of the program: `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 (uma

frase) 'Olá Mundo!!!', toda frase, para ser reconhecida como tal, deve estar entre aspas como veremos logo em seguida nos capítulos subsequentes.

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.

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)	True ou False / o 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 = True    #Booleano (verdadeiro ou falso)
```

```
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!!!"""
```

```
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 = True   #Booleano (verdadeiro ou falso)

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:

Global frame

var_tipo_texto	"Conjunto de caracteres alfanuméricos"
var_tipo_int	12
var_tipo_float	15.3
var_tipo_bool	1
var_tipo_lista	
var_tipo_dicionario	
var_tipo_tupla	
var_tipo_comentario	"Um comentário atribuído a uma variável deixa de ser apenas um comentário, vira frase!!!"

list

0	1	2	3	4
1	2	"Paulo"	"Maria"	True

dict

"nome"	"Fernando"
"idade"	31

tuple

0	1	2
"Ana"	"Fernando"	3

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 à 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 ideia 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:


```
1 nome = 'Maria'
2 #Maria é a nova funcionária
3
4 print('Bem vinda Maria!!!')
5 #acima está uma mensagem de boas vindas a ela.
6
```

```
Bem vinda Maria!!!
```

Exemplo 2:

```
1 '''Este programa está sendo escrito
2 para que Maria, a nova funcionária,
3 comece a se integrar com o sistema.'''
4
```

```
'Este programa está sendo escrito\npara que Maria, a nova funcionária,\na se integrar 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.

```
1 '''Exemplo de comentário
2 | este, inclusive, não será
3 | lido pelo interpretador'''
4
```

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

```
1  comentario1 = '''Exemplo de comentário
2  | | | | | | | | e agora sim será lido
3  | | | | | | | | pelo interpretador!!!'''
4  print(comentario1)
5
```

Exemplo de comentário
e agora sim será lido
pelo interpretador!!!

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.

```
1  var1 = 2019
2  var2 = 2020
3
4  soma = var1 + var2
5
6  print(soma)
7
```

4039

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.

```
1  var1 = 2019
2  var2 = 2020
3  var3 = 2021
4
5  #soma = var1 + var2
6
7  print(var1 + var2)
8
```

4039

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 trazê-la de volta à ativa basta “descomentar” a mesma.

VARIÁVEIS / OBJETOS

Uma variável basicamente é um espaço alocado na memória ao qual iremos armazenar um dado, valor ou informação, simples assim.

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 sempre que necessário 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:

```
1  variavel1 = 11
2
```

Neste caso, inicialmente declaramos uma variável de nome `variavel1` que por sua vez está recebendo o valor 11 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.

```
1  variavel1 = 11
2
3  print(variavel1)
4
```

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 possuímos 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.

```
1  variavel1 = 'Ana'
2  variavel2 = 'Pedro'
3
4  variavel_1 = 'Ana'
5  variavel_2 = 'Pedro'
6
```

Ambos os modelos apresentam sintaxe correta, cabendo ao usuário escolher qual modo ele considera mais fácil de identificar essas variáveis pelo nome.

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

```
1  variavel 1 = 'String'
2  14 = 'Numero'
3

File "<ipython-input-5-47d7a087c1af>", line 1
    variavel 1 = 'String'
        ^
SyntaxError: invalid syntax
```

Nem `variavel 1` nem `a 14` serão reconhecidas pelo interpretador como variáveis porque estão escritas de forma não reconhecida por sua 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:


```
1 NOME = 'Fernando'
2
```

*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:

```
1 NOME = 'Fernando'
2 nome = 'Rafael'
3
```

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

```
1 variável = 'Maria'
2 cômodos = 3
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:

```
1  variavel = 'Ana'
2
3  variavel1 = 'Ana'
4
5  variavel_1 = 'Ana'
6
7  var_num_1 = 'Ana'
8
9  minhavariavel = 'Ana'
10
11 minha_variavel = 'Ana'
12
13 minhaVariavel = 'Ana'
14
```

Permitidos, mas não recomendados:

```
1  variável = 'Ana'
2
3  VARIABEL = 'Ana'
4
5  Variavel = 'Ana'
6
```

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

```
1  1991 = 'Ana'
2
3  minha variavel = 'Ana'
4
5  1variavel = 'Ana'
6
```

*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 usá-las 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 uma variável.

Declarando múltiplas variáveis

É possível, para deixar o código mais enxuto, declarar várias variáveis em uma linha, independentemente do tipo de dado a ser recebido, desde que se respeite a sintaxe correta de acordo com o tipo de dado e sua justaposição.

Por exemplo, o código abaixo:

```
1 nome = 'Maria'
2 idade = 32
3 sexo = 'F'
4 altura = 1.89
5
```

Pode ser agrupado em uma linha, da seguinte forma:

```
1 nome, idade, sexo, altura = 'Maria', 32, 'F', 1.89
2
```

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:

```
1  num1 = 10
2  x = 10
3  a1 = 10
4
```

Método reduzido:

```
1  num1 = x = a1 = 10
2
```

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.

FUNÇÕES BÁSICAS

Funções, falando de forma bastante básica, são linhas ou blocos de códigos aos quais executarão uma determinada ação a partir de nosso código, seguindo suas instruções.

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 específica, 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:

```
1  print('Seja bem vindo!!!')
2
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:

```
1 nome = input('Digite o seu nome: ')\n2 print('Bem Vindo', nome)\n3
```

```
Digite o seu nome: Fernando\nBem Vindo Fernando
```

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, desde que se respeite a justaposição das mesmas. Ex:

```
1 nome = input('Digite o seu nome: ')
2 print('Bem Vindo', nome, '!!!')
3
```

```
Digite o seu nome: Fernando
Bem Vindo Fernando !!!
```

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:

```
1  nome1 = 'Maria'
2  print(nome1)
3
Maria
```

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:

```
1 nome1 = input('Digite o seu nome: ')
2
3 print(nome1)
4
```

Digite o seu nome: Fernando
Fernando

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: Fernando

*supondo é claro, que o usuário digitou Fernando.

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

```
1 nome1 = input('Digite o seu nome: ')
2 print('Seja bem vindo(a) %s' %(nome1))
3
```

```
Digite o seu nome: Fernando
Seja bem vindo(a) Fernando
```

Da mesma forma que fizemos no exemplo anterior, declaramos uma variável `nome1` e por meio da função `input()` pedimos 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 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()`.

```
1 nome1 = input('Digite o seu nome: ')
2
3 print('Seja bem vindo(a) {} !!!'.format(nome1))
4
```

```
Digite o seu nome: Fernando
Seja bem vindo(a) Fernando !!!
```

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().

```
1 nome1 = input('Digite o seu nome: ')
2 msg1 = 'Por favor entre'
3 msg2 = 'Você é o primeiro a chegar.'
4
5 print('Seja bem vindo {}, {}, {}'.format(nome1, msg1, msg2))
6
```

Digite o seu nome: Fernando
Seja bem vindo Fernando, Por favor entre, Você é o primeiro a chegar.

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.

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

print() avançado – Usando de 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 haverá máscaras de substituição a serem trabalhadas, independentemente do tipo de dado.

Dessa forma se 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:

```
1 nome = input('Digite o seu nome: ')
2 ap = input('Digite o número do seu apartamento: ')
3
4 print(f'Seja bem vindo(a) {nome}, morador(a) do ap nº {ap}')
5
```



```
Digite o seu nome: Fernando
Digite o número do seu apartamento: 134
Seja bem vindo(a) Fernando, morador(a) do ap nº 134
```

Supondo que o usuário digitou respectivamente Fernando e 134...

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:

```
1 nome = 'João'
2 dia_vencimento = 10
3 valor_fatura = 149.90
4
5 print(f'Olá, caro {nome},\n Sua fatura vence dia {dia_vencimento} de janeiro')
6 print(f'\n O valor é de R${valor_fatura}.\n Favor pagar até o prazo para evitar multas.')
7
```

Olá, caro João,
Sua fatura vence dia 10 de janeiro

O valor é de R\$149.9.
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,
Sua fatura vence dia 10 de janeiro

O valor é de R\$149.9.

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:

```
1  num1 = 10
2  num2 = 5.2
3  soma = num1 + num2
4
5  print(soma)
6
7  print(f'O resultado é {soma}')
```

15.2
O resultado é 15.2

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:

```
1 num1 = 10
2 num2 = 5.2
3
4 print(f'O resultado é {num1 + num2}')
5 O retorno será: 15.2
6
```

15.2
O resultado é 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.

Outro exemplo, a variável `a` que antes tinha o valor 10 (int) podemos a qualquer momento alterar para 'Maria' (string).

```
1 a = 10
2
3 print(a)
4
```

10

O resultado será: 10

```
1 a = 10
2 a = 'Maria'
3
4 print(a)
5
```

Maria

O resultado será: Maria.

A ordem de leitura por parte do interpretador faz com que o último dado/valor atribuído a variável seja '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. Sendo assim, devemos tomar cuidado quanto a sobrescrever o dado/valor de uma variável.

Por fim, quando estamos usando variáveis 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:

```
1  num1 = 5
2  num2 = 8.2
3  soma = int(num1) + int(num2)
4
5  print(soma)
6
```

13

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:

```
1 num1 = int(5)
2 num2 = float(8.2)
3 soma = num1 + num2
4
5 print(soma)
6
```

13.2

A regra geral diz que qualquer operação entre um int e um float resultará em float.

O retorno 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.

```
1 num1 = 5
2 num2 = 8.2
3 soma = int(num1) + int(num2)
4
5 # Mesmo que:
6 num1 = int(5)
7 num2 = float(8.2)
8 soma = num1 + num2
9
10 print(soma)
11
```

13.2

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

```
1 num1 = int(5)
2 num2 = float(5)
3
4 print(num1)
5 print(num2)
6
```

5
5.0

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.

```
1 num1 = int(5)
2 num2 = int(8.2)
3 soma = num1 + num2
4
5 print(soma)
6
```

13

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:

```
1 num1 = int(5)
2 num2 = int(8.2)
3
4 print(f'O resultado da soma é: {num1 + num2}')
```

```
O resultado da soma é: 13
```

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 mesmas.

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:

```
1  numero = 5
2
3  print(type(numero))
4
<class 'int'>
```

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:

```
1  numero = '5'
2
3  print(type(numero))
4
<class 'str'>
```

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:

```
1  numero = [5]
2
3  print(type(numero))
4
<class 'list'>
```

O retorno será: <class 'list'>

```
1  numero = {5}
2
3  print(type(numero))
4
<class 'set'>
```

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 função de que a notação do dado não é, como esperado, uma chave : valor.

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:


```
1 frase1 = str('Raquel tem 15 anos')
2
3 print(type(frase1))
4
<class 'str'>
```

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:

```
1 frase1 = str('Raquel tem 15 anos, ')
2 frase2 = str('de verdade')
3
4 print(frase1 + frase2)
5
Raquel tem 15 anos, de verdade
```

O resultado será: Raquel tem 15 anos, de verdade.

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

```

1 frase1 = str('Raquel tem ')
2 frase2 = int(15)
3 frase3 = 'de verdade'
4
5 print(frase1 + frase2 + frase3)
6
-----
TypeError                                Traceback (most recent call
<ipython-input-37-3e32395acec3> in <module>()
      3 frase3 = 'de verdade'
      4
----> 5 print(frase1 + frase2 + frase3)

TypeError: must be str, not int

```

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.

```

1 frase1 = str('Raquel tem ')
2 frase2 = str(15)
3 frase3 = ' de verdade'
4
5 print(frase1 + frase2 + frase3)
6
Raquel tem 15 de verdade

```

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.

```
1 print(frase1 + frase2 + frase3)
2
3 # Mesmo que:
4 print(f'{frase1 + frase2 + frase3}')
5
```

Raquel tem 15 de verdade
Raquel tem 15 de verdade

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.

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, independentemente 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 símbolo de igual para igualar operandos, usaremos ele duplicado “ == ”.

Por exemplo:

```
1  salario = 955
2
```

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:

```
1  salario = 955
2  aumento1 = 27
3
4  print(salario + aumento1)
5
```

982

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:

```
1  mensalidade = 229
2  mensalidade = 229 + 10
3
4  print(mensalidade)
5
```

239

O resultado será 239, pois o último valor atribuído a variável `mensalidade`, atualizando a mesma, 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 atribuído a variável, podendo assim, por meio do operador de atribuição, atualizar o dado ou valor declarado inicialmente, por exemplo:

```
1 nome = 'sem nome'
2 idade = 0
3
4 nome = input('Por favor, digite o seu nome: ')
5 idade = input('Digite a sua idade: ')
6
7 print(nome, idade)
8
```

```
Por favor, digite o seu nome: Fernando
Digite a sua idade: 33
Fernando 33
```

Repare que 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:

```
1  variavel1 = 4
2  variavel1 = variavel1 + 5
3
4  # Mesmo que:
5  variavel1 += 5
6
7  print(variavel1)
8
```

14

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:


```
1  variavel1 = 4
2  resultado = variavel1 - 3
3
4  print(resultado)
5
6  # Mesmo que:
7  variavel2 = 4
8  variavel2 -= 3
9
10 print(variavel2)
11
```

```
1
1
```

Nesse caso, o usuário está subtraindo 3 de variavel1. Sendo $4 - 3$:

O resultado será 1.

Atribuição Multiplicativa:

```
1  variavel1 = 4
2  resultado = variavel1 * 2
3
4  print(resultado)
5
6  # Mesmo que:
7  variavel2 = 4
8  variavel2 *= 2
9
10 print(variavel2)
11
```

```
8
8
```

Nesse caso, o usuário está multiplicando o valor de variavel1 por 2. Logo 4×2 :

O resultado será: 8

Atribuição Divisiva:

```
1  variavel1 = 4
2  resultado = variavel1 / 4
3
4  print(resultado)
5
6  # Mesmo que:
7  variavel2 = 4
8  variavel2 /= 4
9
10 print(variavel2)
11
```

1.0
1.0

Nesse caso, o usuário está dividindo o valor de variavel1 por 4. Sendo $4 / 4$.

O resultado será: 1.0

Módulo de (ou resto da divisão de):

```
1  variavel1 = 4
2  resultado = variavel1 % 4
3
4  print(resultado)
5
6  # Mesmo que:
7  variavel2 = 4
8  variavel2 %= 4
9
10 print(variavel2)
11
```

```
0
0
```

Será mostrado apenas o resto da divisão de variavel1 por 4.

O resultado será: 0

Exponenciação:

```
1  variavel1 = 4
2  resultado = variavel1 ** 8
3
4  print(resultado)
5
6  # Mesmo que:
7  variavel2 = 4
8  variavel2 **= 8
9
10 print(variavel2)
11
```

```
65536
65536
```

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:

```
1  variavel1 = 512
2  resultado = variavel1 // 256
3
4  print(resultado)
5
6  # Mesmo que:
7  variavel2 = 512
8  variavel2 //= 256
9
10 print(variavel2)
11
```

```
2
2
```

Neste caso a divisão retornará um número inteiro (ou arredondado). Ex: 512/256.

O resultado será: 2

Operadores aritméticos

Operadores aritméticos, como o nome sugere, são aqueles que usaremos para realizar operações matemáticas em meio a nossos blocos de código.

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 mencionado 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

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

```
12
```

O resultado será: 12

Subtração

```
1 print(12 - 3)
2
```

```
9
```

O resultado será: 9

Multiplicação

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

```
35
```

O resultado será: 35

Divisão

```
1 print(120 / 6)
2
```

```
20.0
```

O resultado será: 20

Operações com mais de 2 operandos

```
1 print(5 + 2 * 7)
2
19
```

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

Operações dentro de operações:

```
1 print((5 + 2) * 7)
2
49
```

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

```
1 print(3 ** 5)
2
243
```

O resultado será 243, ou seja, $3 \times 3 \times 3 \times 3 \times 3$.

Outra operação possível é a de fazer uma divisão que retorne um número inteiro, “arredondado”, através do operador `//`.
Ex:

```
1 print(9.4 // 3)
2
3.0
```

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 %.

```
1 print(10 % 3)
2
1
```

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:


```
1  numero1 = 12
2  numero2 = 3
3
4  print(numero1 + numero2)
5
6  # Mesmo que:
7  print('O resultado da soma é:', numero1 + numero2)
8
9  # Que pode ser aprimorado para:
10 print(f'O resultado da soma é: {numero1 + numero2}')
11
```



```
15
O resultado da soma é: 15
O resultado da soma é: 15
```

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

Exemplos com os demais operadores:

```
1  x = 5
2  y = 8
3  z = 13.2
4
5  print(x + y)
6  print(x - y)
7  print(x ** z)
8  print(z // y)
9  print(z / y)
10
```



```
13
-3
1684240309.400895
1.0
1.65
```

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).

```
1 print(7 != 3)
2
True
```

O retorno será: True

Afinal, 7 é diferente de 3.

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

O retorno será: False

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

Tabela verdade

Tabela verdade AND (E)

Independentemente 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.

$7 \neq 3$ and $3 > 1$ and $6 == 6$ and $8 \geq 9$
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

Independentemente 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 e V = V
F e 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 F e V e F = V

$F \text{ e } F \text{ e } F \text{ e } F \text{ e } F \text{ e } F \text{ e } F \text{ e } F = F$

Tabela Verdade XOR (OU Exclusivo/um ou outro)

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

$V \text{ e } V = F$

$V \text{ e } F = V$

$F \text{ e } V = V$

$F \text{ e } F = F$

Tabela de Operador de Negação (unário)

$\text{not True} = F$

O mesmo que dizer: Se não é verdadeiro então é falso.

$\text{not False} = V$

O mesmo que dizer: Se não é falso então é verdadeiro.

Também visto como:

$\text{not } 0 = \text{True}$

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.

```
1  saldo = 1000
2  salario = 4000
3  despesas = 2967
4
5  meta = saldo > 0 and salario - despesas >= 0.2 * salario
6
7  print(meta)
8
```

True

Analisando a variável meta: Ela verifica se saldo é maior que zero e se salario menos despesas 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) quanto a um determinado elemento constar ou não dentro da mesma. Por Exemplo:

```
1  lista = [1, 2, 3, 'Ana', 'Maria']
2
3  print(2 in lista)
4
True
```

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:

```
1  lista = [1, 2, 3, 'Ana', 'Maria']
2
3  print('Maria' not in lista)
4
False
```

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 (False).

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

```
1 print(3 > 4)
2 #(3 é maior que 4?)
3
False
```

O resultado será False, pois 3 não é maior que 4.

```
1 print(7 >= 3)
2 #(7 é maior ou igual a 3?)
3
True
```

O resultado será True. 7 é maior ou igual a 3, neste caso, maior que 3.

```
1 print(3 >= 3)
2 #(3 é maior ou igual a 3?)
3
True
```

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

Operadores usando variáveis

```
1 x = 2
2 y = 7
3 z = 5
4
5 print(x > z)
6
```

False

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

```
1 x = 2
2 y = 7
3 z = 5
4
5 print(z <= y)
6
```

True

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

```
1 x = 2
2 y = 7
3 z = 5
4
5 print(y != x)
6
```

True

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

Operadores usando condicionais

```
1  if (2 > 1):  
2  | | | print('2 é maior que 1')  
3  
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):

```
1  num1 = 2  
2  num2 = 1  
3  
4  if num1 > num2:  
5  | | print('2 é maior que 1')  
6  
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:

```
1  aluguel = 250
2  energia = 250
3  agua = 65
4
5  print(aluguel is energia)
6

True
```

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

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 abstraíamos ideias 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 estejam programadas dentro do mesmo.

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, prevendo todas as possíveis alternativas.

ifs, elifs e elses

Uma das partes mais legais de programação, sem sombra de dúvidas, é quando finalmente 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 sequencial.

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:

```
1 a = 33
2 b = 34
3 c = 35
4
5 if b > a:
6     print('b é MAIOR que a')
7
```

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.

```
1 a = 33
2 b = 33
3 c = 35
4
5 if b > a:
6     print('b é MAIOR que a')
7 elif b == a:
8     print('b é IGUAL a a')
9
```

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

```
1  a = 33
2  b = 1
3  c = 608
4
5  if b > a:
6      print('b é MAIOR que a')
7  elif b == a:
8      print('b é IGUAL a a')
9  else:
10     print('b é MENOR que a')
11
```

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:

```
1  var1 = 18
2  var2 = 2
3  var3 = 'Maria'
4  var4 = 4
5
6  if var2 > var1:
7      print('A segunda variável é maior que a primeira')
8  elif var2 == 500:
9      print('A segunda variável vale 500')
10 elif var3 == var2:
11     print('A variavel 3 tem o mesmo valor da variavel 2')
12 elif var4 is str('4'):
13     print('A variavel 4 não é do tipo string')
14 else:
15     print('Nenhuma condição é verdadeira')
16
```

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 à vontade alterando o valor das variáveis para verificar que tipos 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:

```
1  a = 34
2  b = 33
3  c = 35
4
5  if a > b and c > a:
6      print('a é maior que b e c é maior que a')
7
```

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

```
1  a = 35
2  b = 34
3  c = 35
4
5  if a > b or a > c:
6      print('a é a variavel maior')
7
```

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:


```
1  nota = int(input('Informe a nota: '))
2
3  if nota >= 9:
4      print('Parabéns, quadro de honra')
5  elif nota >= 7:
6      print('Aprovado')
7  elif nota >= 5:
8      print('Recuperação')
9  else:
10     print('Reprovado')
11
```

```
Informe a nota: 8
Aprovado
```

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:

```
1  var1 = 0
2  var2 = int(input('Digite um número: '))
3
4  if var2 > var1:
5      print('Numero maior que ZERO')
6      if var2 == 1:
7          print('O número digitado foi 1')
8      elif var2 == 2:
9          print('O número digitado foi 2')
10     elif var2 == 3:
11         print('O número digitado foi 3')
12     else:
13         print('O número digitado é maior que 3')
14
15 else:
16     print('Número inválido')
17
```

Digite um número: 2
Numero maior que ZERO
O número digitado foi 2

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, convertido 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 2:

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 caractere que não é um número).

Simulando switch/case

Quem está familiarizado com outras linguagens de programação está acostumado a usar a função Switch 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 dados em forma de dicionário (que veremos em detalhes nos capítulos subsequentes), 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:

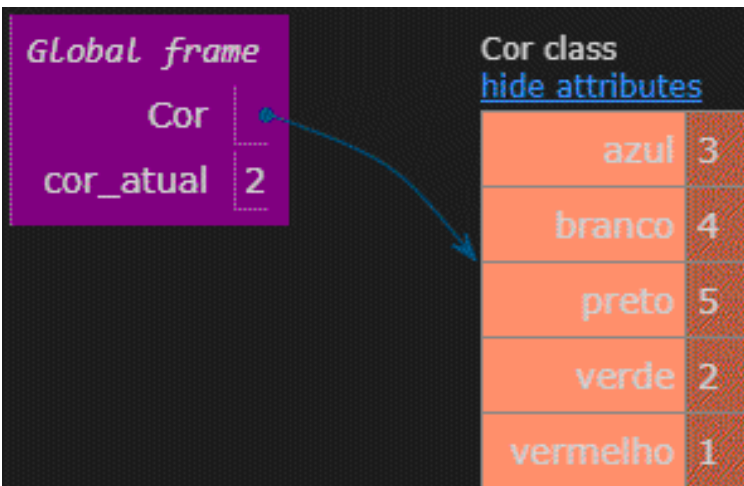
```

1 class Cor:
2     vermelho = 1
3     verde = 2
4     azul = 3
5     branco = 4
6     preto = 5
7
8     # Mude a cor para testar
9     cor_atual = 2
10
11     if cor_atual == Cor.vermelho:
12         print("Vermelho")
13     elif cor_atual == Cor.verde:
14         print("Verde")
15     elif cor_atual == Cor.azul:
16         print("Azul")
17     elif cor_atual == Cor.branco:
18         print("Branco")
19     elif cor_atual == Cor.preto:
20         print("Preto")
21     else:
22         print("Desconhecido")
23

```

Verde

Representação visual:



ESTRUTURAS DE REPETIÇÃO

While

Python tem dois tipos de comandos para executar comandos em loop (executar repetidas vezes uma mesma instrução) o while e o for.

Inicialmente vamos entender como funciona o while. While do inglês, em tradução livre significa enquanto, ou seja, enquanto uma determinada condição for válida, a ação continuará sendo repetida. Por exemplo:

```
1  a = 1
2
3  while a < 8:
4      print(a)
5      a += 1
6
```



```
1
2
3
4
5
6
7
```

Declarada a variável a, de valor inicial 1 (pode ser qualquer valor, inclusive 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:

```
1  a = 1
2
3  while a < 10:
4      print(a)
5      a += 1
6      if a == 4:
7          break
8
1
2
3
```

Enquanto a variável `a` for menor que 10, continue imprimindo-a 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 quisermos 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:

```
1  compras = ['Arroz', 'Feijão', 'Carne', 'Pão']
2
3  for i in compras:
4      |  print(i)
5
```



```
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:

```
1  for x in range(0, 6):
2      print(f'Número {x}')
3
```

Número 0
Número 1
Número 2
Número 3
Número 4
Número 5

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, por meio de nossa função print() podemos ver cada valor atribuído a variável x a cada repetição.

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 fim de um intervalo.

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

```
1  nomes = ['Pedro', 255, 'Leticia']
2
3  for n in nomes:
4      | | print(n)
5
```



```
Pedro
255
Leticia
```

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.

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:

```

1  nomes = ['Pedro', 'João', 'Leticia',]
2
3  for laco in nomes:
4      print(laco)
5  else:
6      print('---Fim da lista!!!---')
7

```

Pedro
João
Leticia
---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:

```
1  for x in range(11):
2      for y in range(11):
3          print(f'{x} x {y} = {x * y}')
4
0 x 5 = 0
0 x 6 = 0
0 x 7 = 0
0 x 8 = 0
0 x 9 = 0
0 x 10 = 0
1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
1 x 10 = 10
2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
```

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

"C:\Users\Fernando\PycharmProjects\Exercici

0 x 0 = 0 1 x 0 = 0 2 x 0 = 0

0 x 1 = 0 1 x 1 = 1 2 x 1 = 2

0 x 2 = 0 1 x 2 = 2 2 x 2 = 4

0 x 3 = 0 1 x 3 = 3 2 x 3 = 6

0 x 4 = 0 1 x 4 = 4 2 x 4 = 8

0 x 5 = 0 1 x 5 = 5 2 x 5 = 10

0 x 6 = 0 1 x 6 = 6 2 x 6 = 12

0 x 7 = 0 1 x 7 = 7 2 x 7 = 14

0 x 8 = 0 1 x 8 = 8 2 x 8 = 16

0 x 9 = 0 1 x 9 = 9 2 x 9 = 18

0 x 10 = 0 1 x 10 = 10 2 x 10 = 20 etc...

STRINGS

Como já vimos algumas vezes ao longo deste livro, 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 caracteres alfanuméricos em geral.

Quando atribuímos 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 atribuído dados do tipo string para uma variável, por exemplo nome = 'Maria', devemos lembrar de o referenciar como tal.

Apenas como exemplo, na sintaxe antiga do Python precisávamos usar do marcador %s em máscaras de substituição para que o interpretador de fato esperasse que aquele dado era do tipo String.

Na sintaxe moderna independente do uso o interpretador lê tudo o que estiver entre aspas ' ' como string.

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:

```
1  'marca d'água'
2

File "<ipython-input-25-a3c48ea86f6a>", line 1
    'marca d'água'
            ^
SyntaxError: invalid syntax
```

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 a apóstrofe) como string.

O mesmo ocorre se invertermos a ordem de uso das aspas. Por exemplo:

```
1  frase1 = 'Era um dia "muuuito" frio'
2
3  print(frase1)
4

Era um dia "muuuito" frio
```

O retorno será: Era um dia "muuuito" frio

```
Global frame
```

```
frase1 "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 atribuímos esse texto de várias linhas a uma variável, ele passa a ser um objeto comum, do tipo string.

```
1  """Hoje tenho treino
2  |   Amanhã tenho aula
3  |   Quinta tenho consulta"""
4
```

*Esse texto nessa forma é apenas um comentário.

```
1  texto1 = """Hoje tenho treino
2  |   Amanhã tenho aula
3  |   Quinta tenho consulta"""
4
5  print(texto1)
6
```

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

*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.

Global frame

texto1	"Hoje tenho treino Amanhã tenho aula Quinta tenho consulta"
--------	---

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

```
1 frase1 = 'A linguagem Python é muito fácil de aprender.'  
2  
3 print(frase1.lower())  
4  
a linguagem python é muito fácil de aprender.
```

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:

```
1 frase1 = 'A linguagem Python é muito fácil de aprender.'  
2  
3 print(frase1.upper())  
4
```

```
A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.
```

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, tratando o mesmo apenas em sua exibição.

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

```
1 frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'  
2 frase1 = frase1.lower()  
3  
4 print(frase1)  
5
```

```
a linguagem python é muito fácil de aprender.
```

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:

```
1 frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
2 frase1 = frase1.lower()
3
4 print('py' in frase1.lower())
5
True
```

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

O 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:

```
1 frase1 = 'A LINGUAGEM PYTHON É MUITO FÁCIL DE APRENDER.'
2 frase1 = frase1.lower()
3
4 print(frase1.split())
5
['a', 'linguagem', 'python', 'é', 'muito', 'fácil', 'de', 'aprender.']
```

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:

```
1 ERRO = '\033[91m'    #código de cores "vermelho"
2 NORMAL = '\033[0m'
3
4 print(ERRO, 'Mensagem de erro aqui' , NORMAL)
5
```

Mensagem de erro aqui.

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).

```
1 ERRO = '\033[91m'    #código de cores "vermelho"
2 NORMAL = '\033[0m'
3
4 print(ERRO + 'Mensagem de erro' + NORMAL)
5
```

Mensagem de erro

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:

```
1 frase1 = 'Bem Vindo ao Meu Programa!!!'
2
3 print(frase1.center(50))
4
```

Bem Vindo ao Meu Programa!!!

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:

```
1 frase1 = 'Bem Vindo ao Meu Programa!!!'
2
3 print(frase1.rjust(50))
4
```

Bem Vindo ao Meu Programa!!!

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:

```
1  from math import pi
2
3  print(f'O número pi é: {pi}')
4
```

O número pi é: 3.141592653589793

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:

```
1  num1 = 34.295927957329247
2
3  print('O valor da ação fechou em %.2f'%num1)
4
```

O valor da ação fechou em 34.30

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”).

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 frequência, uma lista será um objeto que permite guardar diversos dados dentro dele, de forma organizada e indexada.

É como se você 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 rápida do que trabalhar com eles individualmente. Ex:

```
1 lista = [ ]  
2
```

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

```
1 lista2 = [1, 5, 'Maria', 'João']  
2  
3 print(lista2)  
4  
[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:

```
1 lista2 = [1, 5, 'Maria', 'João']
2 lista2.append(4)
3
4 print(lista2)
5
```



```
[1, 5, 'Maria', 'João', 4]
```

Irá adicionar o valor 4 na última posição do índice da lista.

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



Lembrando que por enquanto, esses comandos são sequenciais, ou seja, a cada execução do comando `append()` para inserir um novo elemento na lista, o mesmo será inserido automaticamente na última posição da mesma.

Removendo dados manualmente

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

```
1 lista2 = [1, 5, 'Maria', 'João']
2 lista2.append(4)
3 lista2.remove('Maria')
4
5 print(lista2)
6
```

```
[1, 5, 'João', 4]
```

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

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 próprio 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]`

```
1 lista2 = ['Ana', 'Carlos', 'João', 'Sonia']
2 del lista2[2]
3
4 print(lista2)
5
```



```
['Ana', 'Carlos', 'Sonia']
```

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:

```
1 lista2 = [1, 5, 'Maria', 'João']
2 lista2.append(4)
3
4 print(lista2.index('Maria'))
5
```

2

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

Representação 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 operadores 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.

```
1  lista2 = [1, 5, 'Maria', 'João']
2  lista2.append(4)
3
4  print('João' in lista2)
5
```

```
True
```

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 íntegra). Ex:

```
1 lista1 = []
2
3 lista1.append('Paulo')
4 lista1.append('Maria')
5
6 print(lista1)
7
```

['Paulo', 'Maria']

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:



```
1 lista1 = []
2
3 lista1.append('Paulo')
4 lista1.append('Maria')
5
6 print(lista1)
7
8 lista1.reverse()
9
10 print(lista1)
11
```

['Paulo', 'Maria']
['Maria', 'Paulo']

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:

```
1 lista = [1, 2, 4, 'Paulo']
2
3 print(lista)
4
```

```
[1, 2, 4, 'Paulo']
```

Adicionando uma nova lista no índice 4 da lista atual ficaria:

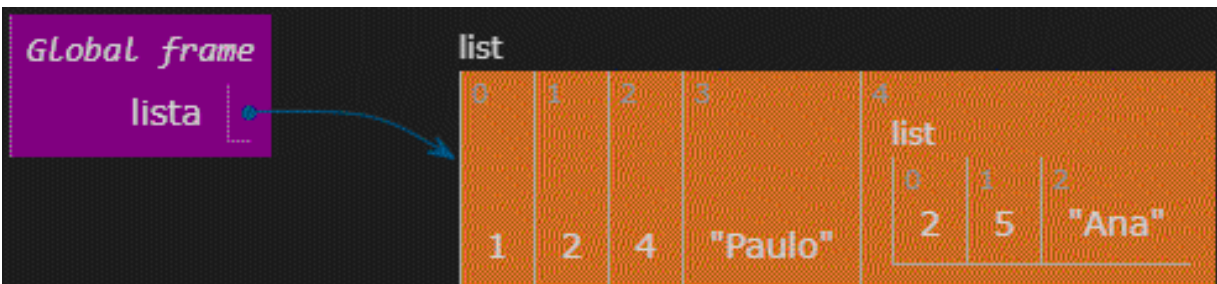
```
1 lista = [1, 2, 4, 'Paulo']
2
3 print(lista)
4
5 lista = [1, 2, 4, 'Paulo', [2, 5, 'Ana']]
6
7 print(lista)
8
```

```
[1, 2, 4, 'Paulo']
```

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

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

Representação visual:



Trabalhando com 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.

```
1  minhaturpla = tuple( )  
2
```

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:

```
1  minhaturpla = (1, )  
2  
3  print(minhaturpla)  
4  
(1,)
```

```
1  minhaturpla = tuple( )
2
3  type(minhaturpla)
4

tuple
```

Se você executar `type(minhaturpla)` 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).

```
1  minhaturpla = (1, )
2
3  dir(minhaturpla)
4
```

```
'__init__',
'__init_subclass__',
'__iter__',
'__le__',
'__len__',
'__lt__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'count',
'index']
```

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

Sendo assim, você pode acessar o índice exatamente como fazia com sua lista. Ex:

```
1  minhaturpla = (1, 2, 3)
2  minhaturpla[0]
3
1
```

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:

```
1  minhaturpla = tuple('Ana')
2  minhaturpla2 = ('Ana',)
3  minhaturpla3 = ('Ana', 'Maria')
4
5  print(minhaturpla)
6  print(minhaturpla2)
7  print(minhaturpla3)
8
'A', 'n', 'a')
'Ana',)
'Ana', 'Maria')
```

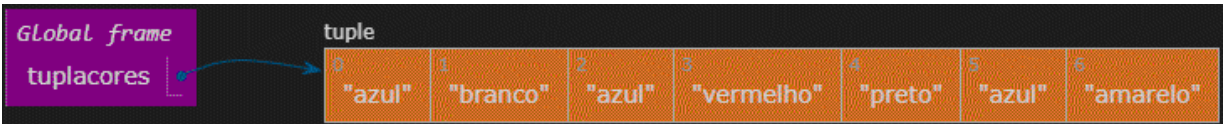
Em minhaturpla existe a atribuição de tipo tuple(), em minhaturpla2 existe a vírgula como separador mesmo não havendo um segundo elemento, em minhaturpla3 já não é mais necessário isto para que o identifique o tipo de dado como tupla.

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

```
1  tuplacoress = ('azul', 'branco', 'roxo', 'azul',  
2  | | | | | | | 'vermelho', 'preto', 'azul', 'amarelo')  
3  
4  tuplacoress.count('azul')  
5  
3
```

Executando o código `tuplacoress.count('azul')`, o retorno será 3 porque existem dentro da tupla 'azul' 3 vezes.

Representação visual:



Trabalhando com 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.

```
1 pilha = [10, 20, 30]
2 pilha.append(50)
3
4 print(pilha)
5
```

```
[10, 20, 30, 50]
```

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.

```
1 pilha = [10, 20, 30]
2 pilha.pop()
3
4 print(pilha)
5
```

```
[10, 20]
```

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:

```
1 pilha = [10, 20, 30]
2 pilha.append(50)
3
4 print(pilha)
5 print(len(pilha))
6
```

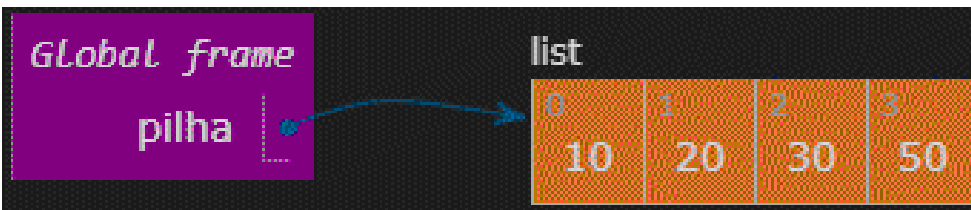
```
[10, 20, 30, 50]
```

```
4
```

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;

DICIONÁRIOS

Enquanto listas e tuplas são estruturas indexadas, um dicionário, além da sintaxe também diferente, se resume em organizar dados em formato chave : valor.

Assim como em um dicionário normal você tem uma palavra e seu respectivo significado, aqui a estrutura lógica será a mesma.

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:

```
1 fernando = {'nome': 'Fernando Feltrin',
2             'idade': 31,
3             'formacao': ['Eng. da Computação', 'Técnico em TI']}
4
5 print(fernando)
6
```

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

Repare que inicialmente o dicionário foi atribuído a um objeto, depois 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.

Global frame

fernando

dict

"nome"	"Fernando Feltrin"						
"idade"	31						
"formacao"	<table><tr><td>list</td></tr><tr><td><table><tr><td>0</td><td>"Engenheiro da Computação"</td></tr><tr><td>1</td><td>"Técnico em TI"</td></tr></table></td></tr></table>	list	<table><tr><td>0</td><td>"Engenheiro da Computação"</td></tr><tr><td>1</td><td>"Técnico em TI"</td></tr></table>	0	"Engenheiro da Computação"	1	"Técnico em TI"
list							
<table><tr><td>0</td><td>"Engenheiro da Computação"</td></tr><tr><td>1</td><td>"Técnico em TI"</td></tr></table>	0	"Engenheiro da Computação"	1	"Técnico em TI"			
0	"Engenheiro da Computação"						
1	"Técnico em TI"						

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.

```
1  fernando = {'nome': 'Fernando Feltrin',  
2             'idade': 31,  
3             'formacao': ['Engenheiro da Computação',  
4                          'Técnico em TI']}  
5
```

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 à quantidade de chaves:valores inclusos no dicionário. Ex:

```
1  fernando = {'nome': 'Fernando Feltrin',  
2             'idade': 31,  
3             'formacao': ['Engenheiro da Computação',  
4                          'Técnico em TI']}  
5  
6  print(len(fernando))  
7  
3
```

O retorno será 3.

Consultando chaves/valores de um dicionário

Você pode consultar o valor de dentro de um dicionário pelo comando `.get()`. Isso será feito consultando uma determinada chave para obter como retorno seu respectivo valor. Ex:

```
1  fernando = {'nome': 'Fernando Feltrin',  
2             'idade': 31,  
3             'formacao': ['Engenheiro da Computação',  
4                          'Técnico em TI']}  
5  
6  print(fernando.get('idade'))  
7  
31
```

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:

```
1 fernando = {'nome': 'Fernando Feltrin',  
2             'idade': 31,  
3             'formacao': ['Engenheiro da Computação',  
4                           'Técnico em TI']}  
5  
6 print(fernando.keys())  
7
```

`dict_keys(['nome', 'idade', 'formacao'])`

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:

```
1 fernando = {'nome': 'Fernando Feltrin',  
2             'idade': 31,  
3             'formacao': ['Engenheiro da Computação',  
4                           'Técnico em TI']}  
5  
6 print(fernando.values())  
7
```

```
dict_values(['Fernando Feltrin', 31, ['Engenheiro da Computação',  
                                     'Técnico em TI']])
```

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:

```
1 fernando = {'nome': 'Fernando Feltrin',
2             'idade': 31,
3             'formacao': ['Engenheiro da Computação',
4                           'Técnico em TI']}
5
6 print(fernando.items())
7
```



```
dict_items([('nome', 'Fernando Feltrin'), ('idade', 31), ('formacao',
('formacao', ['Engenheiro da Computação', 'Técnico em TI'])])
```

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, pré-programados, mas queremos alterá-los, digamos, atualizar os dados dentro de nosso dicionário, podemos fazer isso manualmente através de alguns simples comandos.

Supondo que inicialmente temos um dicionário:

```
1  pessoa = {'nome': 'Alberto Feltrin',
2            'idade': '42',
3            'formação': ['Tec. em Radiologia'],
4            'nacionalidade': 'brasileiro'}
5
6  print(pessoa)
7
```

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

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

Representação visual:




```
1  pessoa = {'nome': 'Alberto Feltrin',
2            'idade': '42',
3            'formação': ['Tec. em Radiologia'],
4            'nacionalidade': 'brasileiro'}
5
6  print(pessoa)
7
8  pessoa['idade'] = 44
9
10 print(pessoa)
11
```

```
{'nome': 'Alberto Feltrin', 'idade': '42', 'formação': ['Tec. em Radiologia'],
 'nacionalidade': 'brasileiro'}
{'nome': 'Alberto Feltrin', 'idade': 44, 'formação': ['Tec. em Radiologia'],
 'nacionalidade': 'brasileiro'}
```

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:

```
1  pessoa = {'nome': 'Alberto Feltrin',
2            'idade': '42',
3            'formação': ['Tec. em Radiologia'],
4            'nacionalidade': 'brasileiro'}
5
6  pessoa['idade'] = 44
7
8  pessoa['formação'].append('Esp. em Tomografia')
9
10 print(pessoa)
11
```

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

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

Representação visual:

Global frame		dict	
pessoa	"nome"	"Alberto Feltrin"	
	"idade"	"42"	
	"formação"	list	
		0	1
		"Tec. em Radiologia"	"Esp. em Tomografia"
	"nacionalidade"	"brasileiro"	

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

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 forma de categorizarmos os números quanto 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 números para poder aplicar funções matemáticas sobre os mesmos.

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:

```
1  a = {1, 2, 3}
2
3  type(a)
4
set
```

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 (apesar da simbologia `{ }`), 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 da função `.union()`, de forma que a operação irá na verdade gerar um terceiro conjunto onde constam os valores dos dois anteriores. Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.union(c2)
5  #c1 união com c2, matematicamente juntar 2 conjuntos.
6
{1, 2, 3}
```

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.

Interseção de conjuntos

Já a interseção entre eles fará o oposto, anulando valores avulsos e mantendo só os que eram comuns aos dois conjuntos. Interseção de dois conjuntos através do operador `.intersection()`.
Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.intersection(c2)
5
{2}
```

O retorno será:{2}, pois este é o único elemento em comum aos dois conjuntos.

Lembrando que quando executamos expressões matemáticas elas apenas alteram o que será mostrado para o usuário, a integridade dos dados iniciais se mantem.

Para alterar os dados iniciais, ou seja, salvar essas alterações, é possível executar o comando `.update()` após as expressões, assim os valores serão alterados de forma permanente.
Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.union(c2)
5  c1.update(c2)
6
7  print(c1)
8
{1, 2, 3}
```

Agora consultando c1 o retorno será {1, 2, 3}

Verificando se um conjunto pertence ao outro

Também é possível verificar se um conjunto está contido dentro do outro, a través da expressão `<=`. Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.union(c2)
5  c1.update(c2)
6
7  c2 <= c1
8
```

```
True
```

O retorno será `True` porque os valores de `c2` (2 e 3) estão contidos dentro de `c1` (que agora é 1, 2 e 3).

Diferença entre conjuntos

Outra expressão matemática bastante comum é fazer a diferença entre dois conjuntos. Ex:

```
1  c1 = {1, 2}
2  c2 = {2, 3}
3
4  c1.union(c2)
5  c1.update(c2)
6
7  c1 = c1 - {2}
8
9  print(c1)
10
```

```
{1, 3}
```

O retorno será: {1, 3} #O elemento 2 foi subtraído dos conjuntos.

INTERPOLAÇÃO


Quando estamos escrevendo nossos primeiros códigos, costumamos escrever todas expressões de forma literal e, na verdade, não há problema nenhum nisto. Porém quando estamos avançando nos estudos de programação iremos ver aos poucos que é possível fazer a mesma coisa de diferentes formas, e a proficiência em uma linguagem de programação se dá quando conseguirmos criar códigos limpos, legíveis, bem interpretados e enxutos em sua sintaxe.

O ponto que eu quero chegar é que pelas boas práticas de programação, posteriormente usaremos uma simbologia em nossas expressões que deixarão nosso código mais limpo, ao mesmo tempo um menor número de caracteres por linha e um menor número de linhas por código resultam em um programa que é executado com performance superior.

Um dos conceitos que iremos trabalhar é o de máscaras de substituição que como o próprio nome já sugere, se dá a uma série de símbolos reservados pelo sistema que servirão para serem substituídos por um determinado dado ao longo do código.

Vamos ao exemplo:

```
1 nome1 = input('Digite o seu nome: ')
2
```



A partir daí poderíamos simplesmente mandar exibir o nome que o usuário digitou e que foi atribuído a variável nome1. Ex:

```
1 nome1 = input('Digite o seu nome: ')
2
3 print(nome1)
4
```

```
Digite o seu nome: Fernando
Fernando
```

O resultado será o nome que o usuário digitou anteriormente. Porém existe uma forma mais elaborada, onde exibiremos uma mensagem e dentro da mensagem haverá uma máscara que será substituída pelo nome digitado pelo usuário.

Vamos supor que o usuário irá digitar “Fernando”, então teremos um código assim:

```
1 nome1 = input('Digite o seu nome: ')
2
3 print('Seja bem vindo %s'%(nome1))
4
```

```
Digite o seu nome: Fernando
Seja bem vindo Fernando
```

O retorno será: Seja bem vindo Fernando

Repare que agora existe uma mensagem de boas vindas seguida da máscara %s (que neste caso está deixando um marcador, que por sua vez identifica ao interpretador que ali será inserido um dado em formato string) e após a frase existe um operador que irá ler o que está atribuído a variável nome1 e irá substituir pela máscara.

As máscaras de substituição podem suportar qualquer tipo de dado desde que referenciados da forma correta: %s(string) ou %d(número inteiro) por exemplo.

Também é possível usar mais de uma máscara se necessário, desde que se respeite a ordem das máscaras com seus

dados a serem substituídos e a sintaxe equivalente, que agora é representada por { } contendo dentro o número da ordem dos índices dos dados a serem substituídos pelo comando .format().

Por exemplo:

```
1  nota1 = '{0} está reprovado, assim como seu colega {1}'  
2  
3  print(nota1.format('Pedro', 'Francisco'))  
4  
  
Pedro está reprovado, assim como seu colega Francisco
```

O Resultado será: Pedro está reprovado, assim como seu colega Francisco

Este tipo de interpolação também funcionará normalmente sem a necessidade de identificar a ordem das máscaras, podendo inclusive deixar as { } chaves em branco.

Avançando com interpolações

```
1 nome = 'Maria'
2 idade = 30
3
```

Se executarmos o comando `print()` da seguinte forma:

```
1 nome = 'Maria'
2 idade = 30
3
4 print('Nome: %s Idade: %d' % (nome, idade))
5
```

Nome: Maria Idade: 30

O interpretador na hora de exibir este conteúdo irá substituir `%s` pela variável do tipo `string` e o `%d` pela variável do tipo `int` declarada anteriormente, sendo assim, o retorno será: Nome: Maria Idade: 30

Python vem sofrendo uma série de "atualizações" e uma delas de maior impacto diz respeito a nova forma de se fazer as interpolações, alguns programadores consideraram essa mudança uma "mudança para pior", mas é tudo uma questão de se acostumar com a nova sintaxe.

Por que estou dizendo isto, porque será bastante comum você pegar exemplos de códigos na web seguindo a sintaxe antiga, assim como códigos já escritos em Python 3 que adotaram essa nova sintaxe que mostrarei abaixo.

Importante salientar que as sintaxes antigas não serão (ao menos por enquanto) descontinuadas, ou seja, você pode escrever usando a que quiser, ambas funcionarão, mas será interessante começar a se acostumar com a nova.

Seguindo o mesmo exemplo anterior, agora em uma sintaxe intermediária:

```
1 nome = 'Maria'
2 idade = 30
3
4 print('Nome: {0} Idade: {1}'.format(nome, idade))
5
```

Nome: Maria Idade: 30

O resultado será: Nome: Maria Idade: 30

Repare que agora o comando vem seguido de um índice ao qual serão substituídos pelos valores encontrados no `.format()`.

E agora por fim seguindo o padrão da nova sintaxe, mais simples e funcional que os anteriores:

```
1 nome = 'Maria'
2 idade = 30
3
4 print(f'Nome: {nome}, Idade: {idade}')
5
```

Nome: Maria, Idade: 30

Repare que o operador `.format()` foi abreviado para simplesmente `f` no início da sentença, e dentro de suas máscaras foram referenciadas as variáveis a serem substituídas.

O resultado será: Nome: Maria Idade: 30

O ponto que eu quero que você observe é que à medida que as linguagens de programação vão se modernizando, elas tendem a adotar sintaxes e comandos mais fáceis de serem lidos, codificados e executados.

A nova sintaxe, utilizando de chaves { } para serem interpoladas, foi um grande avanço também no sentido de que entre chaves é possível escrever expressões (por exemplo uma expressão matemática) que ela será lida e interpretada normalmente, por exemplo:

```
1 nome = 'Maria'
2 idade = 30
3
4 print(f'Nome: {nome}, Idade: {idade}')
5
6 print(f'{nome} tem 3 vezes minha idade, ela tem {3 * idade} anos')
7
```

Nome: Maria, Idade: 30
Maria tem 3 vezes minha idade, ela tem 90 anos

O resultado será: Nome: Maria, Idade: 30

Maria tem 3 vezes minha idade, ela tem 90 anos

Repare que a expressão matemática de multiplicação foi realizada dentro da máscara de substituição, e ela não gerou nenhum erro pois o interpretador está trabalhando com seu valor atribuído, que neste caso era 30.

Representação visual:

Print output

```
Nome: Maria, Idade: 30
Maria tem 3 vezes minha idade, ela tem 90 anos
```

Global frame	
nome	"Maria"
idade	30

FUNÇÕES

Já vimos anteriormente as funções `print()` e `input()`, duas das mais utilizadas quando estamos criando estruturas básicas para nossos programas. Agora que você já passou por outros tópicos e já acumula uma certa bagagem de conhecimento, seria interessante nos aprofundarmos mais no assunto funções, uma vez que existem muitas possibilidades novas quando dominarmos o uso das mesmas, já que o principal propósito é executar certas ações por meio de funções.

Uma função, independentemente da linguagem de programação, é um bloco de códigos que podem ser executados e reutilizados livremente, quantas vezes forem necessárias.

Uma função pode conter ou não parâmetros, que nada mais são do que instruções a serem executadas cada vez que a referenciamos associando as mesmas com variáveis, sem a necessidade de escrever repetidas vezes o mesmo bloco de código da mesma funcionalidade.

Para ficar mais claro imagine que no corpo de nosso código serão feitas diversas vezes a operação de somar dois valores e atribuir seu resultado a uma variável, é possível criar uma vez esta calculadora de soma e definir ela como uma função, de forma que eu posso posteriormente no mesmo código reutilizar ela apenas mudando seus parâmetros.

Funções predefinidas

A linguagem Python já conta com uma série de funções pré definidas e pré carregadas por sua IDE, prontas para uso. Como dito anteriormente, é muito comum utilizarmos, por exemplo, a função `print()` que é uma função dedicada a exibir em tela o que é lhe dado como parâmetro (o que está dentro dos parênteses). Outro exemplo seria a função `len()` que mostra o tamanho de um determinado dado.

Existe a possibilidade de importarmos módulos externos, ou seja, funções já criadas e testadas que estão disponíveis para o usuário, porém fazem parte de bibliotecas externas, que não vêm carregadas por padrão.

Por exemplo é possível importar de uma biblioteca chamada `math` as funções `sin()`, `cos()` e `tg()`, que respectivamente, como seu nome sugere, funções prontas para calcular o seno, o cosseno e a tangente de um determinado valor.

Funções personalizadas

Seguindo o nosso raciocínio, temos funções prontas pré carregadas, funções prontas que simplesmente podemos importar para nosso código e certamente haverá situações onde o meio mais rápido ou prático será criar uma função própria personalizada para uma determinada situação.

Basicamente quando necessitamos criar uma função personalizada, ela começará com a declaração de um `def`, é o meio para que o interpretador assume que a partir dessa palavra reservada está sendo criada uma função.

Uma função personalizada pode ou não receber parâmetros (variáveis / objetos instanciados dentro de parênteses), pode retornar ou não algum dado ou valor ao usuário e por fim, terá um bloco de código indentado para receber suas instruções.

Função simples, sem parâmetros

```
1 def boas_vindas():
2     print('Olá, seja bem vindo ao meu programa!!!')
3     print('Espero que você tenha uma boa experiência...')
4
5 print(boas_vindas())
6
```

Olá, seja bem vindo ao meu programa!!!
Espero que você tenha uma boa experiência...
None

Analisando o código podemos perceber que na primeira linha está o comando `def` seguido do nome dado a nova função, neste caso `boas_vindas()`.

Repare que após o nome existe dois pontos `:` e na linha abaixo, indentado, existem dois comandos `print()` com duas frases.

Por fim, dado o comando `print(boas_vindas())` foi chamada a função e seu conteúdo será exibido em tela.

O retorno será: Olá, seja bem vindo ao meu programa!!!
Espero que você tenha uma
boa experiência...

Representação visual:

Print output

```
Olá, seja bem vindo ao meu programa!!!  
Espero que você tenha uma boa experiência...  
None
```

Global frame

boas_vindas

function

boas_vindas()

boas_vindas

Return
value

None

Como mencionado anteriormente, uma vez definida essa função `boas_vindas()` toda vez que eu quiser usar ela (seu conteúdo na verdade) basta fazer a referência corretamente.

Função composta, com parâmetros

```
1 def eleva_numero_ao_cubo(num):  
2     valor_a_retornar = num * num * num  
3     return(valor_a_retornar)  
4  
5 num = eleva_numero_ao_cubo(5)  
6  
7 print(num)  
8  
125
```

Repare que foi definida a função `eleva_numero_ao_cubo(num)` com `num` como parâmetro, posteriormente foi criada uma variável temporária de nome `valor_a_retornar` que pega `num` e o multiplica por ele mesmo 3 vezes (elevando ao cubo), por fim existe a instrução de retornar o valor atribuído a `valor_a_retornar`.

Seguindo o código existe uma variável `num` que chama a função `eleva_numero_ao_cubo` e dá como parâmetro o valor 5 (que pode ser alterado livremente), e finalizando executa o comando `print()` de `num` que irá executar toda a mecânica criada na função `eleva_numero_ao_cubo` com o parâmetro dado, neste caso 5.

O retorno será: 125

Representação visual:

Global frame

eleva_numero_ao_cubo	
num	125

function

eleva_numero_ao_cubo(num)

eleva_numero_ao_cubo

num	5
valor_a_retornar	125
Return value	125

Função composta, com *args e **kwargs

Outra possibilidade que existe em Python é a de trabalharmos com *args e com **kwargs, que nada mais são do que instruções reservadas ao sistema para que permita o uso de uma quantidade variável de argumentos (parâmetros para uma função) sem necessariamente os declarar de imediato.

Por conversão se usa a nomenclatura args e kwargs, mas na verdade você pode usar o nome que quiser, o marcador que o interpretador buscará na verdade serão os * asteriscos simples ou duplos.

Importante entender também que, como estamos falando em passar um número variável de parâmetros, isto significa que internamente no caso de *args os mesmos serão tratados como uma lista e no caso de **kwargs eles serão tratados como dicionário.

Exemplo *args

```
1  def print_2_vezes(*args):
2      for parametro in args:
3          print(parametro + '!' + parametro + '!')
4
5  print_2_vezes('Olá Mundo!!! ')
6
```

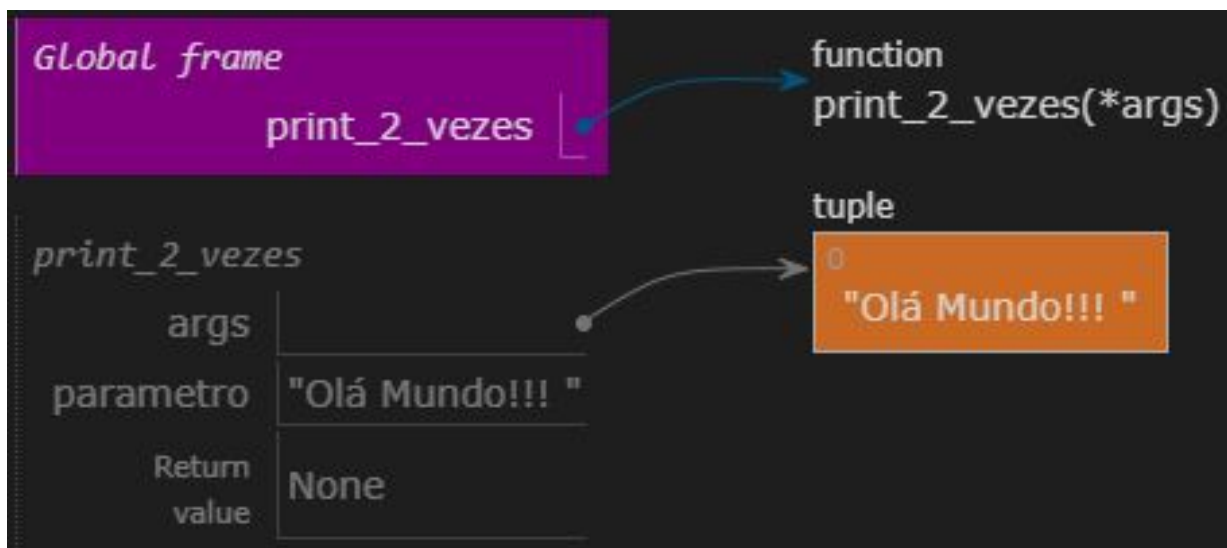
Olá Mundo!!! !Olá Mundo!!! !

Definida a função print_2_vezes(*args) é passado como instrução um comando for, ou seja, um laço que irá verificar se a variável temporária parametro está contida em args, como instrução do laço é passado o comando print, que irá exibir duas vezes e concatenar o que estiver atribuído a variável temporária parametro.

Por fim, seguindo o código é chamada a função `print_2_vezes` e é passado como parâmetro a string 'Olá Mundo!!!'. Lembrando que como existem instruções dentro de instruções é importante ficar atento a indentação das mesmas.

O retorno será: Olá Mundo!!! !Olá Mundo!!!

Representação visual:



Exemplo de ****kwargs**:

```

1  def informacoes(**kwargs):
2      for dado, valor in kwargs.items():
3          print(dado + '-' + str(valor))
4
5  pessoa = informacoes(nome='Fernando',
6                        idade=30,
7                        nacionalidade='Brasileiro')
8
9  print(pessoa)
10

```

```

nome-Fernando
idade-30
nacionalidade-Brasileiro
None

```

Definida a função `informacoes(**kwargs)` é passado como instrução o comando `for`, que irá verificar a presença dos dados atribuídos a `dado` e `valor`, uma vez que `kwargs` espera que, como num dicionário, sejam atribuídas chaves:valores.

Por fim, se estes parâmetros estiverem contidos em `kwargs` será dado o comando `print` dos valores de `dados` e `valores`, convertidos para string e concatenados com o separador “-”.

Então é chamada a função `informações()` pela variável `pessoa` e é passado como chaves e valores para a mesma os dados contidos no código acima.

O retorno será: nome-Fernando

idade-30

nacionalidade-Brasileiro

Representação visual:

Global frame

informacoes

function
informacoes(**kwargs)

informacoes

kwargs

dado

valor

Return

value

"nacionalidade"

"Brasileiro"

None

dict

"nome"

"Fernando"

"idade"

30

"nacionalidade"

"Brasileiro"

dir() e help()

O Python como padrão nos fornece uma série de operadores que facilitam as nossas vidas para executar comandos com nossos códigos, mas é interessante você raciocinar que o que convencionalmente usamos quando estamos codificando nossas ideias não chega a ser 10% do que o python realmente tem para oferecer.

Estudando mais a fundo a documentação ou usando o comando dir, podemos perguntar a um tipo de objeto (int, float, string, list, etc...) todos os recursos internos que ele possui, além disso é possível importar para o python novas funcionalidades, que veremos futuramente neste mesmo curso.

Como exemplo digamos que estamos criando uma lista para guardar nela diversos tipos de dados. Pela sintaxe, uma lista é um objeto/variável que pode receber qualquer nome e tem como característica o uso de colchetes [] para guardar dentro os seus dados. Ex:

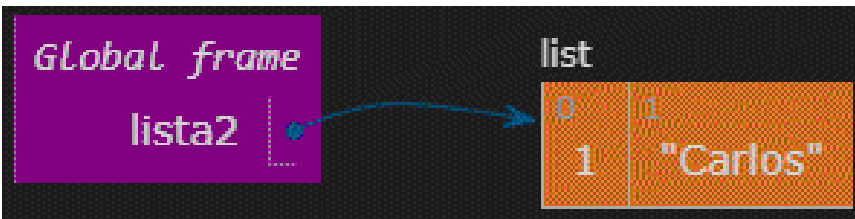
```
1 lista1 = []  
2
```

Lista vazia

```
1 lista2 = [1, 'Carlos']  
2
```

Lista com o valor 1 alocado na posição 0 do índice e com 'Carlos' alocado na posição 1 do índice.

Representação visual:



```
1 lista2 = [1, 'Carlos']
2
3 print(dir(lista2))
4
```

Usando o comando `dir(lista2)` Iremos obter uma lista de todas as possíveis funções que podem ser executadas sobre uma lista

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

```
1 lista2 = [1, 'Carlos']
2
3 print(help(lista2))
4
```

Muito parecido com o `dir()` é o comando `help()`, você pode usar o comando `help(lista2)` e você receberá também uma lista (agora muito mais detalhada) com todas possíveis funções a serem executadas sobre a suas listas. Ex:

O retorno será:

`list(iterable=(), /)`

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.

Methods defined here:

`__add__(self, value, /)`
Return self+value.

`__contains__(self, key, /)`
Return key in self.

`__delitem__(self, key, /)`
Delete self[key].

`__eq__(self, value, /)`
Return self==value.

`__ge__(self, value, /)`
Return self>=value.

`__getattr__(self, name, /)`
Return getattr(self, name).

`__getitem__(...)`
`x.__getitem__(y) <==> x[y]`

`__gt__(self, value, /)`
Return self>value.

`__iadd__(self, value, /)`
Implement self+=value.

`__imul__(self, value, /)`

Implement `self*=value`.

`__init__(self, /, *args, **kwargs)`

Initialize self. See `help(type(self))` for accurate signature.

`__iter__(self, /)`

Implement `iter(self)`.

`__le__(self, value, /)`

Return `self<=value`.

`__len__(self, /)`

Return `len(self)`.

`__lt__(self, value, /)`

Return `self<value`.

`__mul__(self, value, /)`

Return `self*value`.

`__ne__(self, value, /)`

Return `self!=value`.

`__repr__(self, /)`

Return `repr(self)`.

`__reversed__(self, /)`

Return a reverse iterator over the list.

`__rmul__(self, value, /)`

Return `value*self`.

`__setitem__(self, key, value, /)`

Set `self[key]` to value.

`__sizeof__(self, /)`

Return the size of the list in memory, in bytes.

`append(self, object, /)`

Append object to the end of the list.

`clear(self, /)`

Remove all items from list.

`copy(self, /)`

Return a shallow copy of the list.

`count(self, value, /)`

Return number of occurrences of value.

`extend(self, iterable, /)`

Extend list by appending elements from the iterable.

`index(self, value, start=0, stop=9223372036854775807, /)`

Return first index of value.

Raises `ValueError` if the value is not present.

`insert(self, index, object, /)`

Insert object before index.

`pop(self, index=-1, /)`

Remove and return item at index (default last).

Raises `IndexError` if list is empty or index is out of range.

`remove(self, value, /)`

Remove first occurrence of value.

Raises `ValueError` if the value is not present.

`reverse(self, /)`

Reverse **IN PLACE**.

```
| sort(self, /, *, key=None, reverse=False)
|     Stable sort *IN PLACE*.
|
| -----
| Static methods defined here:
|
| __new__(*args, **kwargs) from builtins.type
|     Create and return a new object. See help(type) for accurate
signature.
|
| -----
| Data and other attributes defined here:
|
| __hash__ = None
```

Repare que foi classificado e listado cada recurso possível de ser executado em cima de nossa lista2, assim como o retorno que é esperado em cada uso.

BUILTINS

Builtins, através do comando `dir()`, nada mais é do que uma forma de você pesquisar quais são os módulos e recursos que já vieram pré-alocados em sua IDE. De forma que você pode pesquisar quais são as funcionalidades que estão disponíveis, e, principalmente as indisponíveis, para que você importe o módulo complementar necessário.

Para ficar mais claro, imagine que você consegue fazer qualquer operação matemática básica pois essas funcionalidades já estão carregadas, pré-alocadas e prontas para uso em sua IDE.

Caso você necessite usar expressões matemáticas mais complexas, é necessário importar a biblioteca `math` para que novas funcionalidades sejam inclusas em seu código.

Na prática você pode executar o comando `dir(__builtins__)` para ver tudo o que está disponível em sua IDE em tempo real.

```
1 print(dir(__builtins__))  
2
```

O retorno será:

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',  
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',  
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',  
'ConnectionRefusedError', 'ConnectionResetError',  
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',  
'Exception', 'False', 'FileExistsError', 'FileNotFoundError',  
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',  
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',  
'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',  
'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError',  
'None', 'NotADirectoryError', 'NotImplemented',
```

```
'NotImplementedError',      'OSError',      'OverflowError',
'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError',      'RecursionError', 'ReferenceError',
'ResourceWarning',      'RuntimeError',      'RuntimeWarning',
'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning',
'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True',
'TypeError',      'UnboundLocalError',      'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'WindowsError', 'ZeroDivisionError', '__build_class__', '__debug__',
'__doc__', '__import__', '__loader__', '__name__', '__package__',
'__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'breakpoint',
'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals',
'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min',
'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Cada elemento dessa lista é um módulo pré-alocado, pronta para uso. Importante lembrar que o que faz com que certas palavras e expressões fiquem reservadas ao sistema é justamente um módulo ou biblioteca carregados no builtin.

Não é possível excluir tudo o que está pré-carregado, mas só a fim de exemplo, se você excluísse esses módulos básicos o interpretador não teria parâmetros para verificar no código o que é o que de acordo com a sintaxe.

Repare na sintaxe, `__builtins__` é um tipo de variável, mas ela é reservada do sistema, você não pode alterá-la de forma alguma, assim como outros elementos com prefixo ou sufixo `__`.

Importando bibliotecas

Em Python, de acordo com a nossa necessidade, existe a possibilidade de trabalharmos com as bibliotecas básicas já inclusas, ou importarmos outras bibliotecas que nos tragam novas funcionalidades.

Python como já dissemos anteriormente algumas vezes, é uma linguagem de programação "com pilhas inclusas", ou seja, as bibliotecas básicas que você necessita para grande parte das funções básicas já vem incluídas e pré-alocadas, de forma que basta chamarmos a função que queremos para de fato usá-la.

Porém, dependendo de casos específicos, podemos precisar fazer o uso de funções que não são nativas ou que até mesmo são de bibliotecas externas ao Python padrão, sendo assim necessário que importemos tais bibliotecas para acessarmos suas funções.

Em Python existem duas formas básicas de trabalharmos com as importações, de forma bastante simples, podemos importar uma biblioteca inteira a referenciando pelo nome após o comando `import`, ou podemos importar apenas algo de dentro de uma biblioteca externa para incorporarmos em nosso código, através do comando `from (nome da biblioteca) import (nome da função)`. Por exemplo:

O Python em todas suas versões já conta com funções matemáticas pré-alocadas, porém para usarmos algumas funções ou constantes matemáticas, é necessário importá-las. Supondo que por algum motivo nosso código precise de π , para que façamos o cálculo de alguma coisa. Podemos definir manualmente o valor de π e atribuir a algum objeto, ou podemos importá-lo de alguma biblioteca externa.

No primeiro exemplo, podemos importar a biblioteca math, assim, quando referenciarmos o valor de pi ele já estará pré-alocado para uso, por exemplo.

```
1 import math
2
3 raio = 15.3
4
5 print('Area do circulo', math.pi * raio ** 2)
6
```

Area do circulo 735.4154242788347

O valor de pi, que faz parte da biblioteca math será multiplicado pelo raio e elevado ao quadrado. Como você está importando esse item da biblioteca math, não é necessário especificar o seu valor, neste caso, o valor de pi (3,1416) já está associado a pi.

O resultado será: 735.41

No segundo exemplo, podemos importar apenas a função pi de dentro da biblioteca math, da seguinte forma.

```
1 from math import pi
2
3 raio = 15.3
4
5 print('Area do circulo', pi * raio ** 2)
6
```

Area do circulo 735.4154242788347

O valor de pi será multiplicado pelo raio e elevado ao quadrado.

O retorno será: 735.41

Por fim, apenas para concluir o nosso raciocínio, uma biblioteca é um conjunto de parâmetros e funções já prontas para facilitar a nossa vida, aqui, podemos definir manualmente o valor de pi ou usar ele já pronto de dentro de uma biblioteca.

Apenas faça o seguinte exercício mental, imagine ter que programar manualmente todas as funções de um programa (incluindo as interfaces, entradas e saídas, etc...), seria absolutamente inviável.

Todas linguagens de alto nível já possuem suas funções de forma pré-configuradas de forma que basta incorporá-las ao nosso builtin e fazer o uso.

MÓDULOS E PACOTES

Um módulo, na linguagem Python, equivale ao método de outras linguagens, ou seja, o programa ele executa dentro de um módulo principal e à medida que vamos o codificando, ele pode ser dividido em partes que podem ser inclusive acessadas remotamente.

Um programa bastante simples pode rodar inteiro em um módulo, mas conforme sua complexidade aumenta, e também para facilitar a manutenção do mesmo, ele começa a ser dividido em partes.

Já uma função inteira que você escreve e define, e que está pronta, e você permite que ela seja importada e usada dentro de um programa, ela recebe a nomenclatura de um pacote.

Por padrão, implícito, quando você começa a escrever um programa do zero ele está rodando como modulo `__main__`. Quem vem de outras linguagens de programação já está familiarizado a, quando se criava a estrutura básica de um programa, manualmente criar o método main.

Em Python essa e outras estruturas básicas já estão pré-definidas e funcionando por padrão, inclusive fazendo a resolução dos nomes de forma interna e automática.

Modularização

Uma vez entendido como trabalhamos com funções, sejam elas pré definidas ou personalizadas, hora de entendermos o que significa modularizações em programação.

Apesar do nome assustar um pouco, um módulo nada mais é do que pegarmos blocos de nosso código e o salvar em um arquivo externo, de forma que podemos importar o mesmo posteriormente conforme nossa necessidade.

Raciocine que como visto em capítulos anteriores, uma função é um bloco de código que está ali pré-configurado para ser usado dentro daquele código, mas e se pudéssemos salvá-lo de forma a reutilizar o mesmo em outro código, outro programa até mesmo de um terceiro. Isto é possível simplesmente o transformando em um módulo.

Como já mencionado anteriormente diversas vezes, Python é uma linguagem com pilhas inclusas, e isto significa que ela já nos oferece um ambiente pré-configurado com uma série de bibliotecas e módulos pré carregados e prontos para uso.

Além disto é possível importar novos módulos para adicionar novas funcionalidades a nosso código assim como também é possível criarmos um determinado bloco de código e o exportar para que se torne um módulo de outro código.

Seguindo com nosso raciocínio aqui é onde começaremos a trabalhar mais diretamente importando e usando o conteúdo disponível em arquivos externos, e isto é muito fácil de se fazer desde que da maneira correta.

Em modularização tudo começa com um determinado bloco de código que se tornará um módulo, código pronto e revisado, sem erros e pronto para executar a sua função desejada, salve o mesmo com um nome de fácil acesso e com a extensão .py. Em Python

todo arquivo legível pela IDE recebe inicialmente a extensão reservada ao sistema .py.

Posteriormente compilando o executável de seu programa isto pode ser alterado, mas por hora, os arquivos que estamos trabalhando recebem a extensão .py.

Partindo para prática:

Vamos modularizar o código abaixo:

```
1 def msg_boas_vindas():
2     print('Seja Muito Bem Vindo ao Meu Programa')
3
4 def msg_para_o_usuario(nome):
5     print(f'{nome}, espero que esteja tendo uma boa experiência...')
6
```

Salve este bloco de código com um nome de fácil identificação, por exemplo, boasvindas.py

Representação visual:



Agora vamos importar esse módulo para dentro de outro arquivo de seu código. Em sua IDE, abra ou crie outro arquivo em branco qualquer, em seguida vamos usar o comando import para importar para dentro desse código o nosso módulo previamente salvo boasvindas.py.

```
1 import boasvindas
2
```

Em seguida, importado o módulo, podemos dar comandos usando seu nome seguido de uma instrução, por exemplo ao digitarmos boasvindas e inserirmos um ponto, a IDE irá nos mostrar que comandos podemos usar de dentro dele, no nosso caso,

usaremos por enquanto a primeira opção, que se refere a função `msg_boas_vindas()`.

```
1 import boasvindas
2
3 boasvindas.msg_boas_vindas()
4
```

Se mandarmos rodar o código, o retorno será: Seja Muito Bem Vindo ao Meu Programa

Seguindo com o uso de nosso módulo, temos uma segunda função criada que pode receber uma string como argumento.

Então pela lógica usamos o nome do módulo, “ponto”, nome da função e passamos o argumento. Por exemplo a string ‘Fernando’.

```
1 import boasvindas
2
3 boasvindas.msg_boas_vindas()
4 boasvindas.msg_para_o_usuario('Fernando')
5
```

O Retorno será: Seja Muito Bem Vindo ao Meu Programa

Fernando, espero que esteja tendo uma boa experiência...

Outro exemplo, supondo que estamos criando uma calculadora, onde para obtermos uma performance melhor (no que diz respeito ao uso de memória), separamos cada operação realizada por essa calculadora em um módulo diferente.

Dessa forma, no corpo de nossa aplicação principal podemos nos focar ao código que irá interagir com o usuário e chamar módulos e suas funções enquanto as funções em si só são executadas de forma independente e no momento certo.

Ex: Inicialmente criamos um arquivo de nome `soma.py` que ficará responsável por conter o código da função que irá somar dois

números.

```
1 def soma(num1, num2):  
2     s = int(num1) + int(num2)  
3     return s  
4
```

Repare no código, dentro desse arquivo temos apenas três linhas que contém tudo o que precisamos, mas em outras situações, a ideia de modularizar blocos de código é que eles podem ser bastante extensos dependendo de suas funcionalidades.

Aqui apenas criamos uma função de nome soma() que recebe como parâmetros duas variáveis num1 e num2, internamente a variável temporária s realiza a operação de somar o primeiro número com o segundo e guardar em si esse valor. Por fim, esse valor é retornado para que possa ser reutilizado fora dessa função.

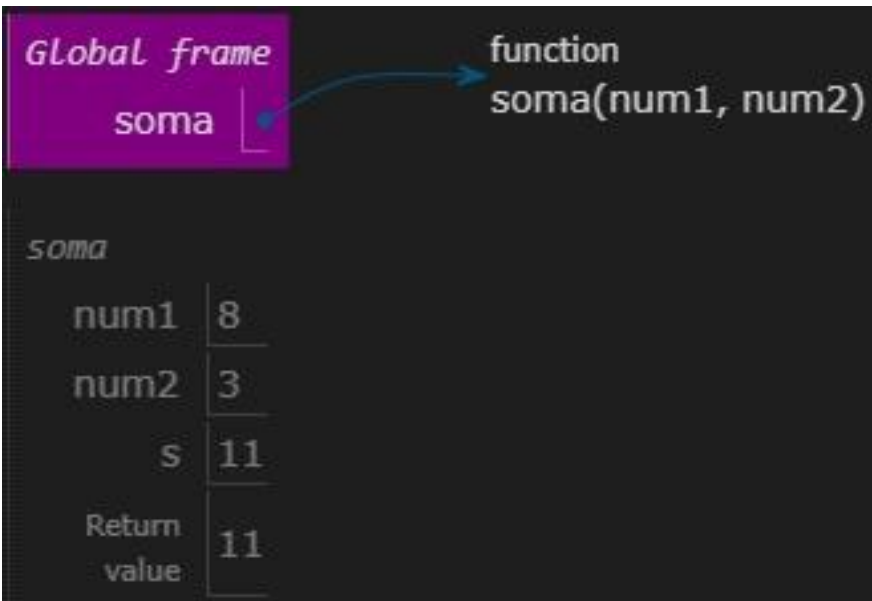
Agora criamos um arquivo de nome index.py e dentro dele o seguinte código:

```
1 import soma  
2  
3 print(f'O resultado da soma é: {soma.soma(8,3)}')  
4
```

Note que inicialmente estamos importando o arquivo soma.py (na importação não é necessário o uso da extensão do arquivo).

Por fim criamos uma mensagem onde na máscara de substituição estamos executando a função soma e passando como parâmetros (para num1 e num2) os números 8 e 3. Nesse caso o retorno será 11.

Representação visual:



Caso você queira criar a interação com o usuário, para nesse exemplo anterior, que o mesmo digite os números, esse processo deve ser feito dentro do módulo da função.

Em `soma.py`

```
1 def soma(num1, num2):
2     num1 = input('Digite um número: ')
3     num2 = input('Digite outro número: ')
4     s = int(num1) + int(num2)
5     return s
6
```

Em `index.py`

```
1 import soma
2
3 print(f'O resultado da soma é: {soma.soma(0,0)}')
4
```

Apenas note que este código é idêntico ao anterior, e na execução do programa, esses valores 0 e 0 definidos aqui serão substituídos e processados pelos valores que o usuário digitar.

Uma dúvida bastante comum a quem está aprendendo a programar em Python é se precisamos usar a extensão do arquivo quando importamos algum módulo e a resposta é não.

Porém, o arquivo de módulo em questão precisa estar na mesma pasta em que estamos trabalhando com os demais arquivos, para que a IDE o ache e consiga o importar.

Na verdade, o Python inicialmente procurará o arquivo em questão no diretório atual, se não encontrar ele buscará o arquivo nos diretórios salvos como PATH. O recomendável é que você mantenha seus arquivos agrupados em um diretório comum.

Outro conceito importante de ser citado a esta altura é o conceito de Pacotes. Em Python não existe uma grande distinção entre esta forma de se trabalhar com módulos se comparado a outras linguagens de programação.

Raciocine que, à medida que você for modularizando seu código, é interessante também dividir categorias de módulos em pastas para melhor organização em geral.

Toda vez que criarmos pastas que guardam módulos dentro de si, em Python costumamos chamar isso de um pacote.

Raciocine também que quanto mais modularizado e organizado é seu código, mais fácil será fazer a manutenção do mesmo ou até mesmo a implementação de novas funcionalidades porque dessa forma, cada funcionalidade está codificada em um bloco independente, e assim, algum arquivo corrompido não prejudica muito o funcionamento de todo o programa.

Por fim, na prática, a grande diferença que haverá quando se trabalha com módulos agrupados por pacotes será a forma com que você irá importar os mesmos para dentro de seu código principal.

Importando de módulos

```
1 import soma
2
```

Importando o módulo soma (supondo que o mesmo é um arquivo de nome soma.py no mesmo diretório).

```
1 import soma as sm
2
```

Importando soma e o referenciando como sm por comodidade.

```
1 from soma import calculo_soma
2
```

Importando somente a função calculo_soma do módulo soma.

Importando de pacotes:

```
1 import calc.calculadora_soma
2
```

Importando calculadora_soma que faz parte do pacote calc. Essa sintaxe calc . calculadora_soma indica que calc é uma subpasta onde se encontra o módulo calculadora_soma.

```
1 from calc.calculadora_soma import soma
2
```


Importando a função soma, do módulo calculadora_soma, do pacote calc.

```
1 from calc.calculadora_soma import soma as sm
2
```

Importando a função soma do módulo calculadora_soma do pacote calc e a referenciando como sm.

Por fim, tenha em mente que todo programa, dependendo claro de sua complexidade, pode possuir incontáveis funcionalidades explícitas ao usuário, mas que o programador definiu manualmente cada uma delas e as categorizou, seguindo uma hierarquia, em funções, módulos, pacotes e bibliotecas.

Você pode fazer o uso de toda essa hierarquia criando a mesma manualmente ou usando de funções/módulos/pacotes e bibliotecas já prontas e pré-configuradas disponíveis para o Python.

O uso de uma biblioteca inteira ou partes específicas dela tem grande impacto na performance de seu programa, pelas boas práticas de programação, você deve ter na versão final e funcional de seu programa apenas aquilo que interessa, modularizar os mesmos é uma prática comum para otimizar o código.

Ex: Criamos um pacote de nome vendas (uma pasta no diretório com esse nome) onde dentro criamos o arquivo calc_preco.py que será um módulo. Na sequência criamos uma função interna para calc_preco. Aqui, apenas para fins de exemplo, uma simples função que pegará um valor e aplicará um aumento percentual.

Em vendas\calc_preco.py

```
1 def aum_preco(preco, porcentagem):
2     npreco = preco + (preco * (porcentagem / 100))
3     return npreco
4
```

Em index.py

```
1 import vendas.calc_preco
2
3 preco = 19.90
4 preco_novo = vendas.calc_preco.aum_preco(preco, 4)
5
6 print(f'O valor corrigido é: {preco_novo}')
7
```

Repare que pela sintaxe, inicialmente importamos o módulo `calc_preco` do pacote `vendas`. Em seguida, como já fizemos outras vezes em outros exemplos, usamos essa função passando parâmetros (um valor inicial e um aumento, nesse caso, de 4%) e por fim exibimos em tela o resultado dessa função.

Poderíamos otimizar esse código da seguinte forma:

```
1 from vendas.calc_preco import aum_preco as ap
2
3 preco = 19.90
4 preco_novo = ap(preco, 4)
5
6 print(f'O valor corrigido é: {preco_novo}')
7
```

O resultado seria o mesmo, com um ganho de performance por parte do processamento. Nesse exemplo, o retorno seria: 20.69.

Representação visual:

<i>Global frame</i>	
aum_preco	
preco	19.9
preco_novo	20.696

<i>aum_preco</i>	
preco	19.9
porcentagem	4
npreco	20.696
Return value	20.696

function
aum_preco(preco, porcentagem)

Apenas a nível de curiosidade, se você consultar a documentação do Python verá uma infinidade de códigos prontos e pré-configurados para uso.

Dependendo o uso, Python tem à disposição bibliotecas inteiras, que contém dentro de si inúmeros pacotes, com inúmeros módulos, com inúmeras funções a disposição, bastando as importar da maneira correta e incorporar em seu código.

**PROGRAMAÇÃO
OBJETOS**

ORIENTADA

A

Classes

Antes de mais nada é importante salientar aqui que, em outras linguagens de programação quando começamos a nos aprofundar nos estudos de classes normalmente há uma separação desde tipo de conteúdo dos demais por estar entrando na área comumente chamada de orientação a objetos

Em Python não há necessidade de fazer tal distinção uma vez que toda a linguagem em sua forma já é nativa orientada a objetos, logo, progressivamente fomos avançando em seus conceitos e avançaremos muito mais sem a necessidade dessa divisão.

Apenas entenda que classes estão diretamente ligadas a programação orientada a objeto, que nada mais é uma abstração de como lidaremos com certos tipos de problemas em nosso código, criando e usando objetos de uma forma mais complexa, a partir de estruturas “moldes”.

Uma classe dentro das linguagens de programação nada mais é do que um objeto que ficará reservado ao sistema tanto para indexação quanto para uso de sua estrutura, é como se criássemos uma espécie de molde de onde podemos criar uma série de objetos a partir desse molde.

Assim como podemos inserir diversos outros objetos dentro deles de forma que fiquem instanciáveis (de forma que permita manipular seu conteúdo), modularizados, oferecendo uma complexa, mas muito eficiente maneira de se trabalhar com objetos.

Parece confuso, mas na prática é relativamente simples, tenha em mente que para Python toda variável é um objeto, a forma como lhe instanciamos e/ou irá fazer com que o interpretador o trate com mais ou menos privilégios dentro do código.

O mesmo acontece quando trabalharmos com classes. Porém o fato de haver este tipo de dado específico “classe” o define como uma estrutura lógica modelável e reutilizável para nosso código.

Uma classe fará com que uma variável se torne de uma categoria reservada ao sistema para que possamos atribuir dados, valores ou parâmetros de maior complexidade, é como se transformássemos uma simples variável em uma super variável, de maior possibilidade de recursos e de uso.

Muito cuidado para não confundir as coisas, o que é comum de acontecer no seu primeiro contato com programação orientada a objetos, raciocine de forma direta que uma classe será uma estrutura molde que poderá comportar dados dentro de si assim como servir de molde para criação de novas variáveis externas.

Pela sintaxe convencionalmente usamos o comando class (palavra reservada ao sistema) para especificar que a partir deste ponto estamos trabalhando com este tipo de dado, na sequência definimos um nome para essa classe, onde por convenção, é dado um nome qualquer desde que o mesmo se inicie com letra maiúscula.

Por exemplo:

```
1 class Carro:
2     carro1 = 'Gol modelo 2016 completo'
3     carro2 = 'Celta ano 2015 4 portas'
4     carro3 = 'Uno ano 2015 baixa quilometragem'
5     carro4 = 'Clio 2018 flex doc vencido'
6
7     print(Carro.carro1)
8
```

Gol modelo 2016 completo

O Retorno será: Gol modelo 2016 completo

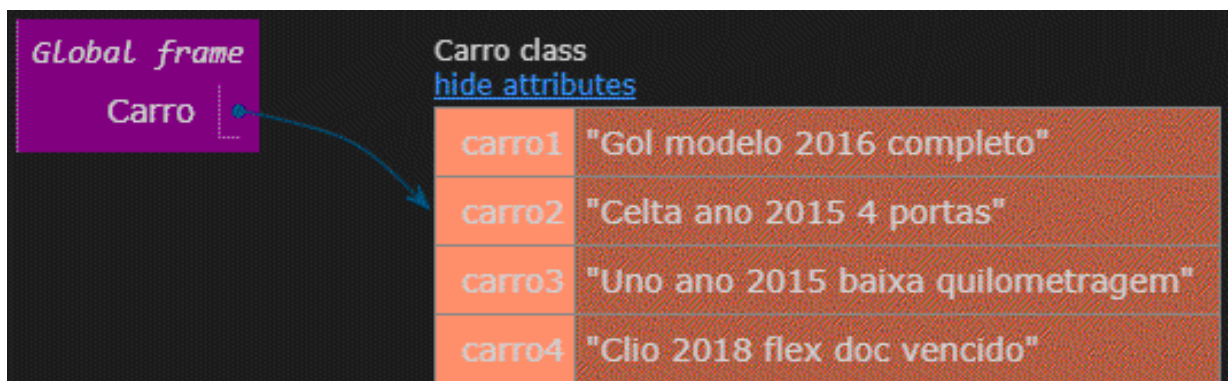
Inicialmente foi criada a classe Carro (class é uma palavra reservada ao sistema e o nome da classe deve ser case sensitive, ou seja, iniciar com letra maiúscula), dentro dessa classe foram inseridos 4 carros com suas respectivas características.

Por fim, o comando `print()` mandou exibir em tela a instância `carro1` que pertence a `Carro`. Seguindo a lógica do conceito explicado anteriormente, `Carro` é uma super variável que tem `carro1` (uma simples variável) contida nele.

O prefixo `class` faz com que o interpretador reconheça essa hierarquia de variáveis e trabalhe da maneira correta com elas.

Abstraindo esse exemplo para simplificar seu entendimento, `Carro` é uma classe/categoria/tipo de objeto, dentro dessa categoria existem diversos tipos de carros, cada um com seu nome e uma descrição relacionada ao seu modelo.

Representação visual:



Para entendermos de forma prática o conceito de uma “super variável”, podemos raciocinar vendo o que ela tem de diferente de uma simples variável contendo dados de qualquer tipo.

Raciocine que pelo básico de criação de variáveis, quando precisávamos criar, por exemplo, uma variável pessoa que dentro de si, como atributos, guardasse dados como nome, idade, sexo, etc...

isto era feito transformando esses dados em uma lista ou dicionário, ou até mesmo criando várias variáveis uma para cada tipo de dado, o que na prática, em programas de maior complexidade, não é nada eficiente.

Criando uma classe Pessoa, poderíamos da mesma forma ter nossa variável pessoa, mas agora instanciando cada atributo como objeto da classe Pessoa, teríamos acesso direto a esses dados posteriormente, ao invés de buscar dados de índice de um dicionário por exemplo.

Dessa forma, instanciar, referenciar, manipular esses dados a partir de uma classe também se torna mais direto e eficiente a partir do momento que você domina essa técnica.

Ex:

```
1 class Pessoa:
2     pass
3
4 pessoa1 = Pessoa()
5
6 pessoa1.nome = 'Fernando'
7 pessoa1.idade = 32
8
9 print(pessoa1.nome)
10 print(pessoa1.idade)
11
```

Fernando
32

Apenas iniciando o entendimento desse exemplo, inicialmente definimos a classe Pessoa que por sua vez está vazia de argumentos. Em seguida criamos a variável pessoa1 que recebe como atribuição a classe Pessoa().

A partir desse ponto, podemos começar a inserir dados (atributos) que ficarão guardados na estrutura dessa classe. Simplesmente aplicando sobre a variável o comando pessoa1.nome

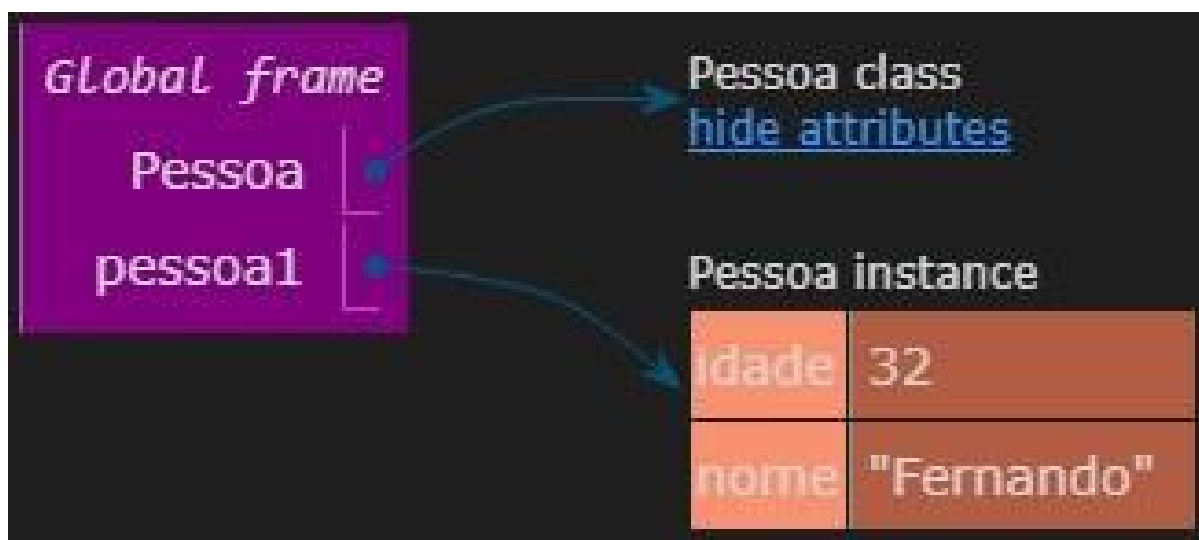
= 'Fernando' estamos criando dentro dessa variável pessoa1 a variável nome que tem como atributo a string 'Fernando'.

Da mesma forma, dentro da variável pessoa1 é criada a variável idade = 32. Note que pessoa1 é uma super variável por conter dentro de si outras variáveis com diferentes dados/valores/atributos... Por fim, passando como parâmetro para nossa função print() pessoa1.nome e pessoa1.idade.

O retorno será: Fernando

32

Representação visual:



No exemplo anterior, bastante básico, a classe em si estava vazia, o que não é o convencional, mas ali era somente para exemplificar a lógica de guardar variáveis e seus respectivos atributos dentro de uma classe.

Normalmente a lógica de se usar classes é justamente que elas guardem dados e se necessário executem funções a partir dos mesmos. Apenas para entender o básico sobre esse processo, analise o bloco de código seguinte:

```
1 class Pessoa:
2     def acao1(self):
3         print('Ação 1 está sendo executada...')
4
5 pessoa1 = Pessoa()
6
7 pessoa1.acao1()
8
```

Ação 1 está sendo executada...

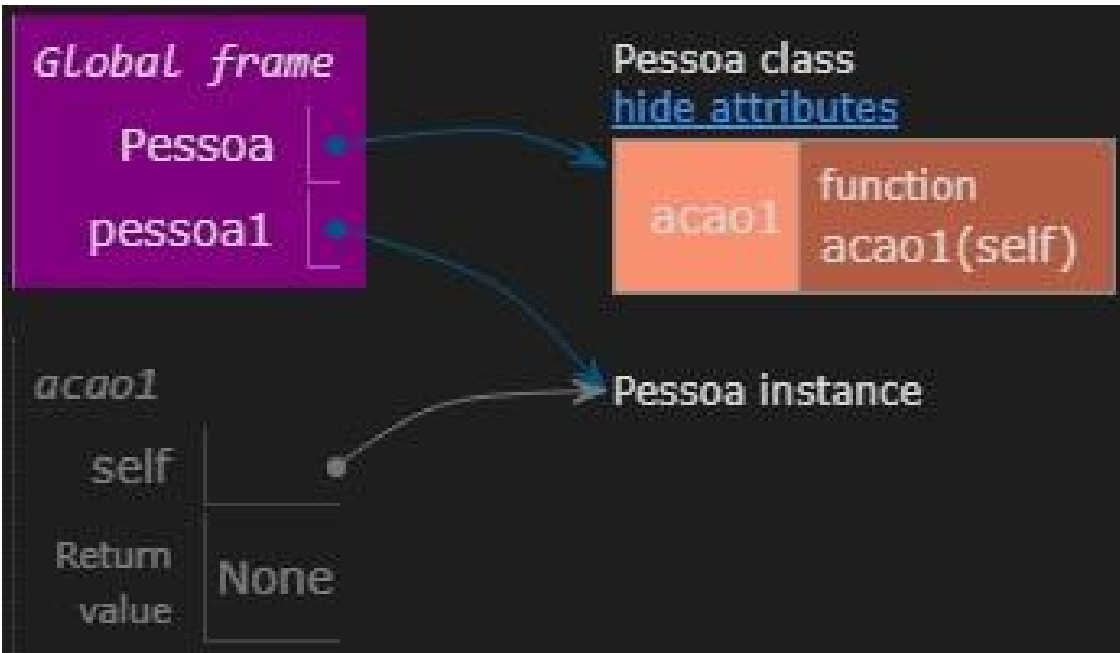
Criada a classe Pessoa, dentro dela é definida a função `acao1(self)` que por sua vez exibe em tela uma mensagem. Na sequência é criada a variável `pessoa1` que tem como atribuição tudo o que estiver dentro da classe Pessoa.

Assim como no exemplo anterior adicionamos variáveis com atributos dentro dessa classe a partir da variável inicial, podemos também chamar uma função interna da classe sobre essa variável diretamente aplicando sobre si o nome da função, nesse caso `pessoa1.acao1()` executará a função interna na respectiva variável.

O retorno será: Ação 1 está sendo executada...

Por fim, note que para tal feito simplesmente referenciamos a variável `pessoa1` e chamamos diretamente a função `.acao1()` sobre ela.

Representação visual:



Definindo uma classe

Em Python podemos manualmente definir uma classe respeitando sua sintaxe adequada, seguindo a mesma lógica de sempre, há uma maneira correta de identificar tal tipo de objeto para que o interpretador o reconheça e trabalhe com ele.

Basicamente quando temos uma função dentro de uma classe ela é chamada de método dessa classe, que pode executar qualquer coisa.

Outro ponto é que quando definimos uma classe manualmente começamos criando um construtor para ela, uma função que ficará reservada ao sistema e será chamada sempre que uma instância dessa classe for criada/usada para que o interpretador crie e use a instância da variável que usa essa classe corretamente.

Partindo para prática, vamos criar do zero uma classe chamada Usuario:

```
1 class Usuario:
2     def __init__(self, nome, idade):
3         self.nome = nome
4         self.idade = idade
5
6     def boas_vindas(self):
7         print(f'Usuário: {self.nome}, Idade: {self.idade}')
8
9 usuario1 = Usuario(nome='Fernando', idade='31')
10 usuario1.boas_vindas()
11
12 print(usuario1.nome)
13
```

Usuário: Fernando, Idade: 31
Fernando

Repare que o código começa com a palavra reservada `class` seguida do nome da classe que estamos definindo, `Usuario`, que pela sintaxe o nome de uma classe começa com letra maiúscula.

Em seguida é declarado e definido o construtor da classe `__init__`, que sempre terá `self` como parâmetro (instância padrão), seguido de quantos parâmetros personalizados o usuário criar, desde que separados por vírgula.

Pela indentação existem duas chamadas de `self` para atribuir um nome a variável `nome` e uma idade a variável `idade`.

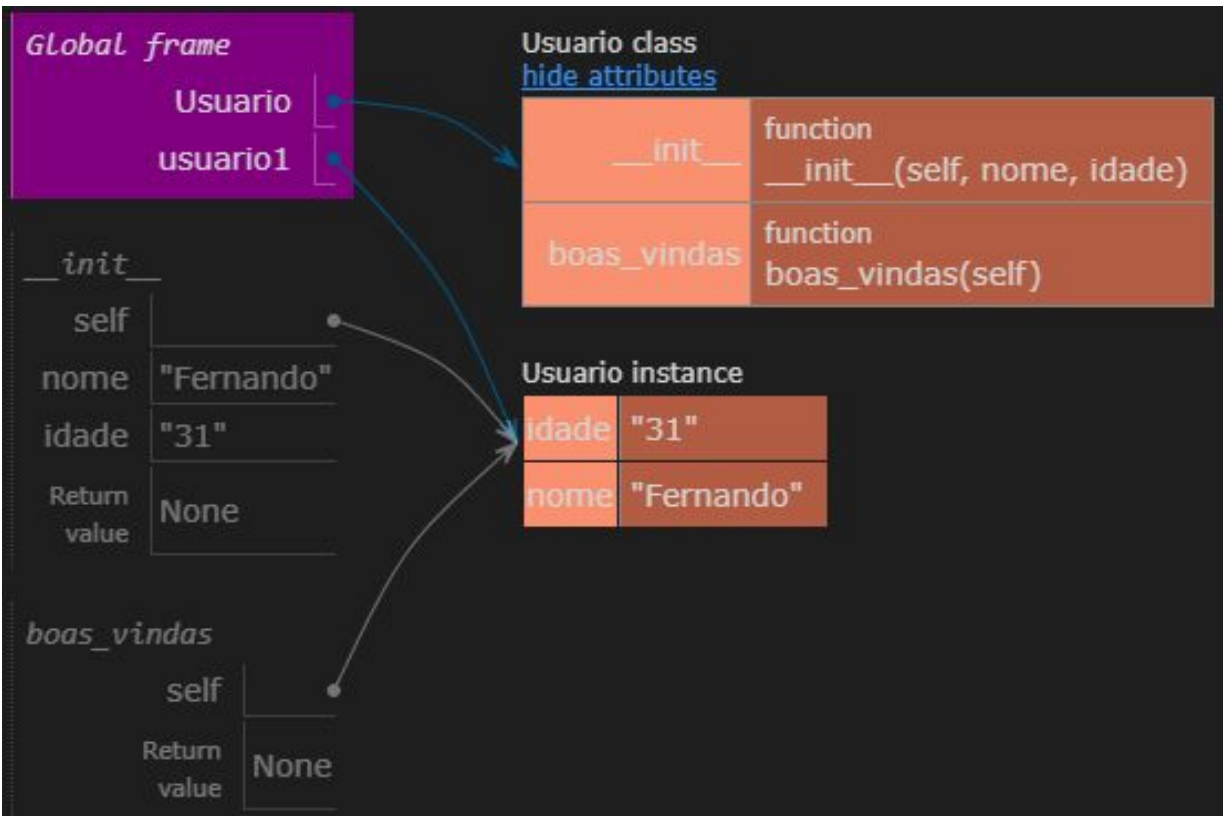
Na sequência há uma nova função apenas chamando a função `print` para uma string que interage com as variáveis temporárias declaradas anteriormente. Por fim é criada uma variável `usuario1` que recebe a classe `Usuário` e a atribui dados para `nome` e `idade`.

Na execução de `usuario1.boas_vindas()` tudo construído e atribuído até o momento finalmente irá gerar um retorno ao usuário. Há ainda uma linha adicional que apenas chama a função `boas_vindas` para a variável `usuário1`, apenas por convenção mesmo.

O retorno será: Usuário: Fernando, Idade: 31

Fernando

Representação visual:



Alterando dados/valores de uma instância

Muito bem, mas e se em algum momento você precisa alterar algum dado ou valor de algo instanciado dentro de uma classe, isto é possível? Sim e de forma bastante fácil, por meio de atribuição direta ou por intermédio de funções que são específicas para manipulação de objetos de classe.

Exemplo de manipulação direta:

```
1  usuario1 = Usuario(nome='Fernando', idade='31')
2  usuario1.nome = 'Fernando Feltrin'
3  usuario1.boas_vindas()
4
5  print(usuario1.nome)
6
```

Usuário: Fernando Feltrin, Idade: 31
Fernando Feltrin

Aproveitando um trecho do código anterior, repare que na terceira linha é chamada a variável `usuario1` seguida de `.nome` atribuindo uma nova string 'Fernando Feltrin'. Ao rodar novamente este código o retorno será: Usuário: Fernando Feltrin, Idade: 31

Exemplo de manipulação via função específica:

```
1  usuario1 = Usuario(nome='Fernando', idade='31')
2  usuario1.nome = 'Fernando Feltrin'
3
4  setattr(usuario1, 'nome', 'Fernando B. Feltrin')
5  delattr(usuario1, 'idade',)
6  setattr(usuario1, 'idade', '32')
7
8  print(usuario1.nome)
9  print(usuario1.idade)
10
```

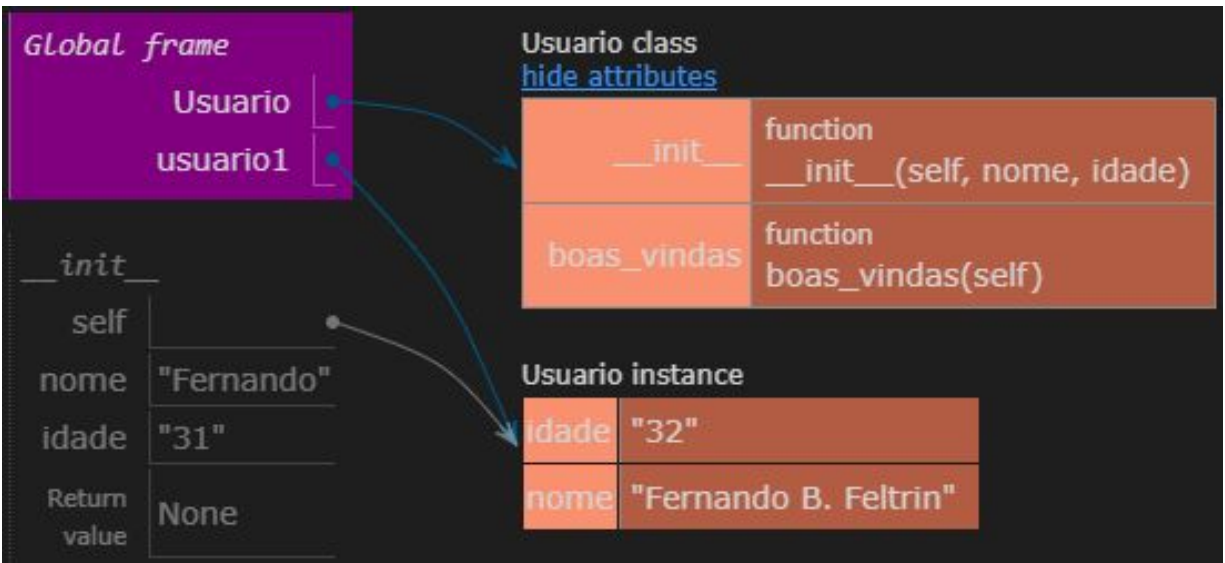
```
Fernando B. Feltrin
32
```

Repare que estamos trabalhando ainda no mesmo trecho de código anterior, porém agora na terceira linha temos a função `setattr()` que tem como parâmetro a variável `usuario1` e irá pegar sua instância `nome` e alterar para nova string 'Fernando B. Feltrin'.

Logo em seguida existe a função `delattr()` que pega a variável `usuario1` e deleta o que houver de dado sob a instância `idade`, logo na sequência uma nova chamada da função `setattr()` irá pegar novamente a variável `usuario1`, agora na instância `idade` e irá atribuir o novo valor, '32', em forma de string mesmo.

Por fim, dando os respectivos comandos `print` para que sejam exibidas as instâncias `nome` e `idade` da variável `usuario1` o retorno serão os valores atualizados. Neste caso o retorno será: Usuário Fernando B. Feltrin, Idade: 32

Representação visual:



Em suma, quando estamos trabalhando com classes existirão funções específicas para que se adicionem novos dados (função `setattr(variavel, 'instancia', 'novo dado')`), assim como para excluir um determinado dado interno de uma classe (função `delattr(variavel, 'instancia')`).

Aplicando recursividade

O termo recursividade em linguagens de programação diz respeito à quando um determinado objeto ou função chamar ele mesmo dentro da execução de um bloco de código executando sobre si um incremento.

Imagine que você tem um objeto com dois parâmetros, você aplicará uma condição a ser atingida, porém para deixar seu código mais enxuto você passará funções que retornam e executam no próprio objeto. Por exemplo:

```
1 def imprimir(maximo, atual):
2     if atual >= maximo:
3         return
4     print(atual)
5     imprimir(maximo, atual + 1)
6
7 imprimir(10, 1)
8
```

```
1
2
3
4
5
6
7
8
9
```

O resultado será: 1

2

3

4

5

6

7

8

9

Repare que o que foi feito é que definimos um objeto imprimir onde como parâmetros (variáveis) temos maximo e atual, em seguida colocamos a condição de que se atual for maior ou igual a maximo, pare a execução do código através de seu retorno. Se essa condição não for atingida, imprima atual, em seguida chame novamente o objeto imprimir, mudando seu segundo parâmetro para atual + 1.

Por fim, definimos que imprimir recebe como argumentos (o que iremos atribuir a suas variáveis) 10 para maximo e 1 para atual. Enquanto atual não for maior ou igual a 10 ele ficará repetindo a execução desse objeto o incrementando em 1.

Em outras palavras, estamos usando aquele conceito de incremento, visto em for anteriormente, mas aplicado a objetos.

Existe a possibilidade de deixar esse código ainda mais enxuto em sua recursividade, da seguinte forma:

```
1 def imprimir(maximo, atual):
2     if atual < maximo:
3         print(atual)
4         imprimir(maximo, atual + 1)
5
6 imprimir(10, 1)
7
```

1
2
3
4
5
6
7
8
9

O Retorno será: 1

2
3
4
5
6
7
8
9

Representação visual:

Global frame
imprimir

function
imprimir(maximo, atual)

imprimir

maximo	10
atual	1
Return value	None

imprimir

maximo	10
atual	5
Return value	None

imprimir

maximo	10
atual	2
Return value	None

imprimir

maximo	10
atual	6
Return value	None

imprimir

maximo	10
atual	3
Return value	None

imprimir

maximo	10
atual	7
Return value	None

imprimir

maximo	10
atual	4
Return value	None

imprimir

maximo	10
atual	8
Return value	None

imprimir

maximo	10
atual	9
Return value	None

Herança

Basicamente quando falamos em herança, a lógica é exatamente a mesma de uma herança “na vida real”, se sou herdeiro de meus pais, automaticamente eu herdo certas características e comportamentos dos mesmos.

Em programação a herança funciona da mesma forma, temos um objeto com capacidade de herdar certos parametros de outro, na verdade uma classe que herda métodos, parâmetros e outras propriedades de sua classe referência.

Imagine que você tem um objeto Carro com uma série de características (modelo, ano, álcool ou gasolina, manual ou automático), a partir dele é criado um segundo objeto Civic, que é um Carro inclusive com algumas ou todas suas características. Dessa forma temos objetos que herdam características de objetos anteriores.

Em Python, como já vimos anteriormente, a estrutura básica de um programa já é pré configurada e pronta quando iniciamos um novo projeto em nossa IDE, fica subentendido que ao começar um projeto do zero estamos trabalhando em cima do método principal do mesmo, ou método `__main__`.

Em outras linguagens inclusive é necessário criar tal método e importar bibliotecas para ele manualmente. Por fim quando criamos uma classe que será herdada ou que herdará características, começamos a trabalhar com o método `__init__` que poderá, por exemplo, rodar em cima de `__main__` sem substituir suas características, mas sim adicionar novos atributos específicos.

Na prática, imagine que temos uma classe Carros, e como herdeira dela teremos duas novas classes chamadas Gol e Corsa, que pela sintaxe deve ser montada com a seguinte estrutura lógica:

```

1  class Carros():
2      def __init__(self, nome, cor):
3          self.nome = nome
4          self.cor = cor
5
6      def descricao(self):
7          print(f'O carro: {self.nome} é {self.cor}')
8
9  class Gol(Carros):
10     def __init__(self, nome, cor):
11         super().__init__(nome, cor)
12
13 class Corsa(Carros):
14     def __int__(self, nome ,cor):
15         super().__init__(nome, cor)
16

```

Repare que primeiro é declarada a classe Carros, sem parâmetro nenhum, porém com um bloco de código indentado, ou seja, que pertence a ele.

Dentro deste bloco existe o construtor com os tipos de dados aos quais farão parte dessa classe, neste caso, iremos atribuir informações de nome e de cor a esta classe.

Há um segundo método definido onde é uma declaração de descrição, onde será executada a impressão de uma string que no corpo de sua sentença irá substituir as máscaras pelas informações de nome e cor que forem repassadas.

Então é criada a classe Gol que tem como parâmetro Carros, sendo assim, Gol é uma classe filha de Carros.

Na linha de código indentada existe o construtor e a função `super()` que é responsável por herdar as informações contidas na classe mãe. O mesmo é feito com a classe filha Corsa.

A partir disto, de toda essa estrutura montada, é possível criar variáveis que agora sim atribuirão dados de nome e cor para as classes anteriormente criadas.


```

1  class Carros():
2      def __init__(self, nome, cor):
3          self.nome = nome
4          self.cor = cor
5
6      def descricao(self):
7          print(f'O carro: {self.nome} é {self.cor}')
8
9  class Gol(Carros):
10     def __init__(self, nome, cor):
11         super().__init__(nome, cor)
12
13     class Corsa(Carros):
14         def __int__(self, nome ,cor):
15             super().__init__(nome, cor)
16
17     gol1 = Gol('Gol 2019 Completo', 'branco')
18     corsa2 = Corsa('Corsa 2017 2 Portas', 'vermelho')
19
20     print(gol1.descricao())
21     print(corsa2.descricao())
22

```

```

O carro: Gol 2019 Completo é branco
None
O carro: Corsa 2017 2 Portas é vermelho

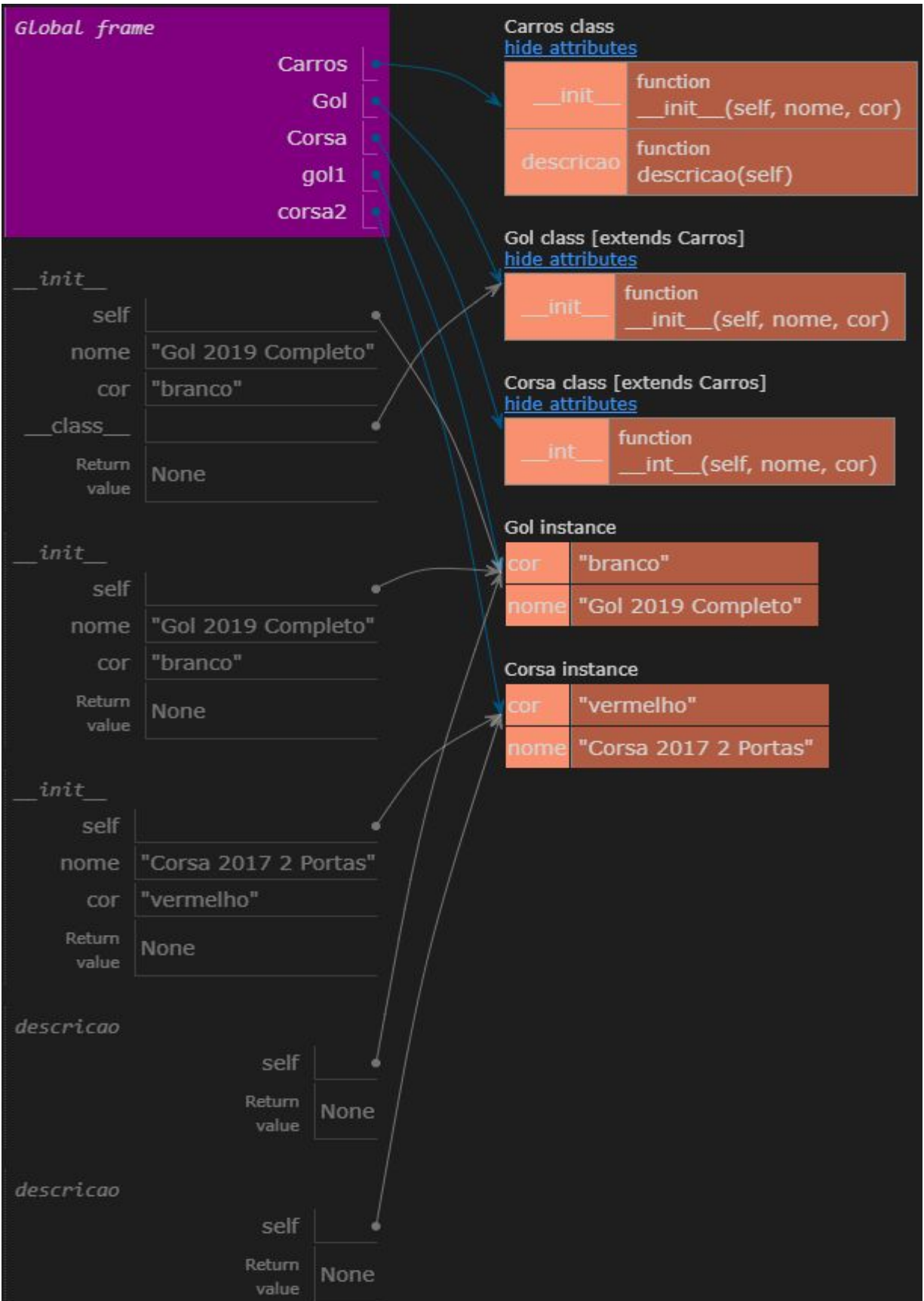
```

Repare que é criada a variável gol1 que recebe a classe Gol e passa como parâmetro duas strings, que pela sequência serão substituídas por nome e cor na estrutura da classe. O mesmo é feito criando a variável corsa2 e atribuindo parametros a classe Corsa.

Por fim é dado o comando print, que recebe como parâmetro a variável gol1 seguido da função definida descricao. gol1 receberá os dados atribuídos a classe Gol e .descricao irá executar sua função print, substituindo as devidas máscaras em sua sentença.

O retorno será: O carro: Gol 2019 Completo é branco

O carro: Corsa 2017 2 Portas é vermelho



Em suma, é possível trabalhar livremente com classes que herdam características de outras, claro que estruturas de tamanha complexidade apenas serão usadas de acordo com a necessidade de seu código possuir de trabalhar com a manipulação de tais tipos de dados.

Polimorfismo

Polimorfismo basicamente é a capacidade de você reconhecer e reaproveitar as funcionalidades de um objeto para uma situação adversa, se um objeto que você já possui já tem as características que você necessita pra quê você iria criar um novo objeto com tais características do zero.

Na verdade, já fizemos polimorfismo no capítulo anterior enquanto usávamos a função `super()` que é dedicada a possibilitar de forma simples que possamos sobrescrever ou estender métodos de uma classe para outra conforme a necessidade.

```
1 class Carros():
2     def __init__(self, nome, cor):
3         self.nome = nome
4         self.cor = cor
5
6 class Gol(Carros):
7     def __init__(self, nome, cor):
8         super().__init__(nome, cor)
9
```

Repare que a classe filha Gol possui seu método construtor e logo em seguida usa da função `super()` para buscar de Carros os parâmetros que ela não tem.

Encapsulamento

O conceito de encapsulamento basicamente é que existe como um objeto e suas funcionalidades ficar encapsulado, fechado de forma que eu consiga instanciá-los sem mesmo conhecer seu conteúdo interior e também é uma forma de tornar um objeto privado, reservado ao sistema de forma que possamos usar livremente porém se o criamos e encapsulamos a ideia é que ele realmente seja imutável.

Principalmente quando usarmos um código de terceiros, muitas das vezes teremos o conhecimento daquele objeto, sua funcionalidade e conseguiremos incorporá-lo em seu código sem a necessidade de alterar ou configurar algo de seu conteúdo, apenas para uso mesmo.

Imagine uma cápsula de um determinado remédio, você em grande parte das vezes desconhece a maioria dos componentes químicos que estão ali dentro, porém você sabe o princípio ativo (para que serve) aquele remédio e o toma conforme a necessidade.

Porém, outro fator importante é que em outras linguagens de programação este conceito de tornar-se um objeto privado é muito levado a sério, de forma que em certos casos ele é realmente imutável.

Em Python, como ela é uma linguagem dinamicamente tipada, na verdade não existirão objetos com atributos privados ao sistema, até porque não há necessidade disso, você ter o controle sempre é importante, e quando necessário, bastará transformar um objeto em um `__objeto__` para que o mesmo fique reservado ao sistema e ainda seja possível o modificar ou incrementar caso necessário.

```
1 objeto1 = 'Descrição por extenso'
2 #variável/objeto de uso comum.
3
4 __objeto1__ = 'Descrição por extenso'
5 #variável/objeto que está reservado ao sistema.
6
7 print(__objeto1__)
8
```

```
Descrição por extenso
```

O retorno será: Descrição por extenso

Como em Python tudo é objeto, tudo é dinâmico, e a linguagem coloca o controle total em suas mãos, há a convenção de alguns autores de que o encapsulamento em Python seria mais um aspecto estético (ao bater o olho em qualquer underline duplo __ saber que ali é algo reservado ao sistema) do que de fato ter de se preocupar com o acesso e a manipulação daquele tipo de variável/objeto, dado ou arquivo.

TRACEBACKS / EXCEÇÕES

Uma das situações que ainda não havíamos comentado, mas que certamente já ocorreu com você ao longo de seus estudos foi o fato de que em certas circunstâncias, quando houver algum erro de lógica ou de sintaxe, o interpretador irá gerar um código de erro.

Tais códigos em nossa IDE são chamados de `tracebacks` e eles tem a finalidade de tentar apontar ao usuário qual é o erro e em que linha do código o mesmo está ocorrendo.

Na prática grande parte das vezes um `traceback` será um erro genérico que apenas irá nos informar o erro, mas não sua solução.

Partindo para camada de software que o usuário tem acesso, nada pior do que ele tentar executar uma determinada função de seu programa e o mesmo apresentar algum erro e até mesmo travar e fechar sozinho. Lembre-se que sempre será um erro humano, de lógica ou de sintaxe.

Por exemplo:


```
1 num1 = 13
2 num2 = ad
3
4 soma = num1 + num2
5
6 print(soma)
7
```

```
NameError                                Traceback (most recent call
<ipython-input-137-85e1e38a52fa> in <module>()
      1 num1 = 13
----> 2 num2 = ad
      3
      4 soma = num1 + num2
      5

NameError: name 'ad' is not defined
```

O retorno será:

Traceback (most recent call last):

File "C:/Users/Fernando/teste_001.py", line 2, in <module> num2 =
ad

NameError: name 'ad' is not defined

Repare no código, declaradas duas variáveis num1 e num2 e uma terceira que faz a soma das duas anteriores, executado o comando print o retorno será um traceback.

Analisando o traceback ele nos mostra que na execução no nosso atual arquivo (no exemplo teste_001.py), na linha 2 o dado/valor ad não é reconhecido como nenhum tipo de dado.

Comandos try, except e finally

Pelas boas práticas de programação, uma solução elegante é prevermos os possíveis erros e/ou na pior das hipóteses apenas mostrar alguma mensagem de erro ao usuário, apontando que está havendo alguma exceção, algo não previsto durante a execução do programa.

Ainda trabalhando em cima do exemplo anterior, o traceback se deu pelo fato de estarmos tentando somar um int de valor 13 e um dado ad que não faz sentido algum para o interpretador.

Através do comando try (em tradução livre do inglês = tentar) podemos fazer, por exemplo, com que o interpretador tente executar a soma dos valores daquelas variáveis.

Se não for possível, será executado o comando except (em tradução livre = exceção), que terá um print mostrando ao usuário uma mensagem de erro e caso for possível realizar a operação o mesmo executará um comando finally (em tradução livre = finalmente) com o retorno e resultado previsto.

```
1
2  try:
3      num1 = int(input('Digite o primeiro numero: '))
4      num2 = int(input('Digite o segundo numero: '))
5
6  except:
7      print('Numero invalido, tente novamente;')
8
9  finally:
10     soma = int(num1) + int(num2)
11     print(f'O resultado da soma é: {soma}')
12
```

Repare que o comando inicial deste bloco de código é o try, pela indentação, note que é declarada a variavel num1 que pede

para o usuário que digite o primeiro número, em seguida é declarada uma segunda variável de nome `num2` e novamente se pede para que o usuário digite o segundo número a ser somado.

Como já vimos anteriormente, o comando `input` aceita qualquer coisa que o usuário digitar, de qualquer tamanho, inclusive com espaços e comandos especiais, o `input` encerra sua fase de captação quando o usuário finalmente aperta ENTER.

Supondo que o usuário digitou nas duas vezes que lhe foi solicitado um número, o código irá executar o bloco `finally`, que por sua vez cria a variável temporária `soma`, faz a devida soma de `num1` e `num2` e por fim exibe em tela uma string com o resultado da soma.

Mas caso ainda no bloco `try` o usuário digitar algo que não é um número, tornando impossível a soma dos mesmos, isto irá gerar uma exceção, o bloco `except` é responsável por capturar esta exceção, ver que algo do bloco anterior está errado, e por fim, neste caso, exibe uma mensagem de erro previamente declarada.

Este é um exemplo de calculadora de soma de dois números onde podemos presumir que os erros que ocorrerão são justamente quando o usuário digitar algo fora da normalidade.

Importante salientar que se você olhar a documentação do Python em sua versão 3 você verá que existem vários muitos tipos de erro que você pode esperar em seu código e por fins de performance (apenas por curiosidade, Python 3 oferece reconhecimento a 30 tipos de erro possíveis), junto do comando `except`: você poderia declarar o tipo de erro para poupar processamento.

Supondo que é um programa onde o tipo de arquivo pode gerar uma exceção, o ideal, pelas boas práticas de programação seria declarar um `except TypeError`: assim o interpretador sabe que é previsto aquele tipo de erro em questão (de tipo) e não testa todos os outros possíveis erros.

Porém em seus primeiros programas não há problema nenhum em usar um `except`: que de forma básica chama esta função que irá esperar qualquer tipo de erro dentro de seu banco de dados sintático.

Na prática você verá que é bastante comum usarmos `try` e `except` em operações onde o programa interage com o usuário, uma vez que é ele que pode inserir um dado inválido no programa gerando erro.

CONSIDERAÇÕES FINAIS

Muito bem, como tudo o que tem um começo tem um fim, e chegamos ao final deste pequeno livro sobre Python. Espero que a leitura deste livro tenha sido tão prazerosa para você quanto foi para mim escrevê-lo. E mais importante do que isso, espero que de fato você tenha aprendido a dar os seus primeiros passos dentro dessa linguagem de programação que é incrível.

Seja por hobby ou para fins profissionais, lembre-se que este é apenas o passo inicial de seu aprendizado de Python, há um mundo de possibilidades esperando por você dentro de todas as áreas da programação em que você pode se especializar.

Agradeço a compra deste material e lhe desejo sucesso em suas novas empreitadas.

Sem mais.

Fernando Feltrin

LIVROS



[Python do ZERO à Programação Orientada a Objetos](#)

[Programação Orientada a Objetos com Python](#)

[Ciência de Dados e Aprendizado de Máquina](#)

[Inteligência Artificial com Python](#)

[Redes Neurais Artificiais com Python](#)

[Análise Financeira com Python](#)

[Arrays com Python + Numpy](#)

[Tópicos Avançados em Python](#)

[Visão Computacional em Python](#)

[Python na Prática \(Exercícios resolvidos e comentados\)](#)

[Tópicos Especiais em Python vol. 1](#)

[Tópicos Especiais em Python vol. 2](#)

[Blockchain e Criptomoedas em Python](#)

[Coletânea Tópicos Especiais em Python](#)

[Python na Prática \(Códigos comentados\)](#)

[PYTHON TOTAL \(Coletânea de 12 livros\)](#)

CURSO

Desenvolvimento > Linguagens de programação > Python

Python do ZERO à Programação Orientada a Objetos

Aprenda programação em Python de forma rápida e efetiva.

Mais bem cotados 4,7 ★★★★★ (79 classificações) 1.445 alunos

Criado por [Fernando Belomé Feltrin](#)

Última atualização em 10/2020 Português Português [Automático]

[Lista de Favoritos](#) [Compartilhar](#) [Presentear este curso](#)

Fernando Belomé Feltrin
Professor



★ 4,7 Classificação do instrutor
👤 79 Avaliações
👥 1.445 Alunos
🎓 1 Cursos

4.7
★★★★★
Classificação do Curso

★★★★★	68%
★★★★☆	25%
★★★☆☆	5%
★★☆☆☆	1%
★☆☆☆☆	1%

Python do ZERO à Programação Orientada a Objetos
Aprenda programação em Python de forma rápida e efetiva.
Fernando Belomé Feltrin
4,7 ★★★★★ (79)
15,5 horas no total • 340 aulas • Iniciante
Classificação mais alta

Pré-visualizar este curso

R\$ [redacted] R\$ [redacted]
38% de desconto
🕒 **Só mais 5 horas** por este preço!

[Adicionar ao carrinho](#)

[Comprar agora](#)

Garantia de devolução do dinheiro em 30 dias

Este curso inclui:

- 📺 15,5 horas de vídeo sob demanda
- 📄 2 artigos
- ∞ Acesso total vitalício
- 📱 Acesso no dispositivo móvel e na TV
- 📜 Certificado de Conclusão

[Aplicar cupom](#)

[Curso Python do ZERO à Programação Orientada a Objetos](#)

Mais de 15 horas de videoaulas que lhe ensinarão programação em linguagem Python de forma simples, prática e objetiva.

REDES SOCIAIS



Prof. Fernando Feltrin - Python

@fernandofeltrinpython · Site educacional

WhatsApp

<https://www.facebook.com/fernandofeltrinpython>

Overview Repositories 4 Projects Packages

Popular repositories

Visao-Computacional
Exemplos do livro VISÃO COMPUTACIONAL EM PYTHON - FERNANDO FELTRIN
☆ 14 🍴 5

Redes-Neurais-Artificiais
Exemplos utilizados nos livros Ciência de Dados e Aprendizado de Máquina / Redes Neurais Artificiais de minha autoria.
Python ☆ 7 🍴 5

Python
Jupyter Notebook

Analise-Financeira-Com-Python

Fernando Belomé Feltrin
fernandofeltrin

<https://github.com/fernandofeltrin>