Programação Básica com Python

Aprendendo programação básica com Python

PROJETO DESENVOLVE

Sumário

Introdução 4

Apresentação 5
Objetivos do curso 6
O que é um algoritmo? 12
Do algoritmo ao código 19
Conhecendo as Ferramentas 22
Conhecendo o Google Colab 30

Fundamentos 33
O que é Variável 33
Tipos de dados e primeiros erros 36
Instruções e expressões 43
Funções básicas: input e print 50

Condicionais 59
Expressões Relacionais 61
Expressões Lógicas 63
Tabela Verdade 67
Condicionais 69
Decisões Múltiplas 73

Estruturas de Repetição 76
Comando while 79
Comando for 85
Comandos Break e continue 90

Escopo e Funções 93

Funções embutidas e bibliotecas 93
Funções Customizadas 99
Escopo de Váriaveis 105
Funções Lambda 107

Listas 111

Propriedades de listas 112
Operações e funções de listas 116
Fatiamento de listas 122
Compreensão de listas 125
Map e Filter 129

Strings e Arquivos 135

Introdução a strings 135
Operações e funções de strings 145
Acessando arquivos 150
Arquivos de texto 157
Arquivos Binários 165

Outras estruturas 167

Set 168
Tupla 176
Dicionários 182

Introdução

Bem-vindo à emocionante jornada de aprendizado da programação! Este material foi cuidadosamente elaborado para introduzir jovens entusiastas como você ao fascinante mundo da programação. Ao longo deste curso, exploraremos os fundamentos da codificação, fornecendo as habilidades necessárias para transformar suas ideias criativas em realidade.

Apresentação

Apresentação

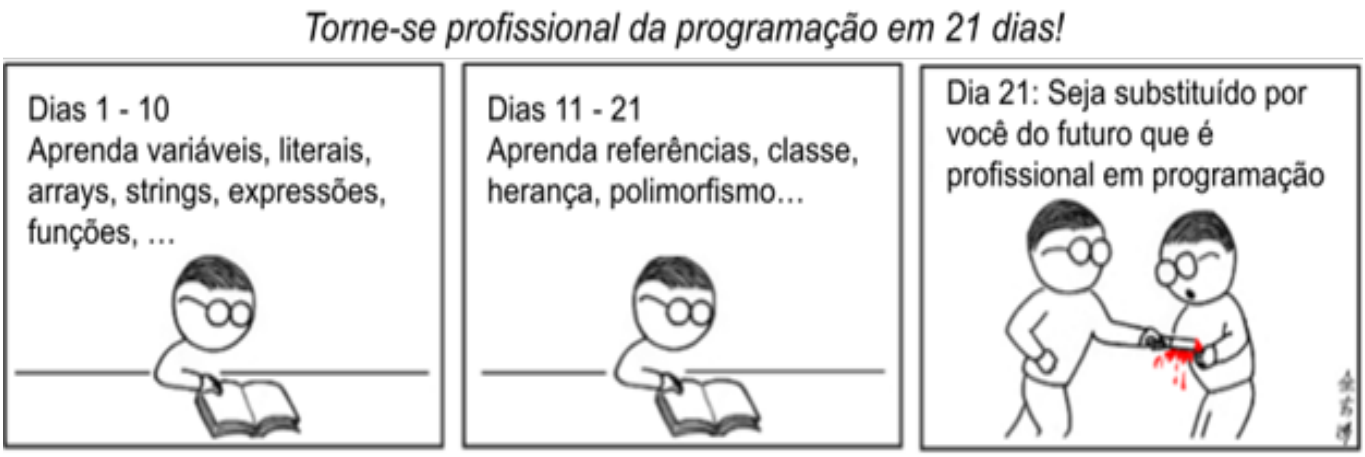
Antes de começar nosso curso de Programação Básica com Python, quero me apresentar brevemente deixando alguns links e dicas que podem facilitar a sua jornada. Me chamo Camila Laranjeira, mas pode me chamar de Mila. Já ministrei disciplinas de Introdução a Programação na UFMG e na PUC-MG para estudantes de 1º período, onde comumente se tem o primeiro contato com programação. Minha motivação é desfazer o mito da genialidade, mostrando que tecnologia pode (e deve!) ser desenvolvida por todas as pessoas que assim desejarem.

·Você pode conhecer mais sobre o meu trabalho e entrar em contato comigo se quiser:
·Minha página acadêmica, meu trabalho com pesquisa e docência
•Youtube do Peixe Babel, meu trabalho com divulgação científica :)
·Email: mila.laranjeira@gmail.com

Neste curso você vai aprender do zero as bases da programação, então não é esperado nenhum conhecimento prévio. Vamos cuidadosamente introduzir cada conceito, desde a definição de algoritmos, passando por cada estrutura de controle e de dados, lhe dando as ferramentas necessárias para desenvolver não só em Python, mas nas inúmeras linguagens que usam essas mesmas estruturas.

·Principais referências do nosso curso:
·Livros: Python para Informática (capítulos 1 a 10), uma adaptação do igualmente importante Pense como Cientista da Computação (capítulos 1 a 11).
·Tutoriais do Real Python. Dica: Você pode traduzir páginas web para português em qualquer navegador. Veja o tutorial do Canal Tech.

E o mais importante: não tenha pressa! O aprendizado é um processo gradual e flui em uma velocidade diferente para cada pessoa. Não caia no conto do "torne-se profissional em 21 dias", isso não produz nada além de frustração. A tirinha a seguir é uma adaptação dessa aqui.



Retomando nossa analogia, assim como diferentes veículos trafegam nas estradas---carros, caminhões, motos---diferentes tipos de pacotes de informações trafegam pela rede da Internet. Alguns pacotes são como caminhões carregando grandes volumes de dados, como arquivos de vídeo ou downloads de software. Outros são como carros de corrida, levando informações rapidamente, como transmissões ao vivo em jogos de computador. A Web é apenas mais um serviço que utiliza essa infraestrutura para trafegar um tipo específico de informação: páginas web.

Objetivos do curso

Da série que você assiste, aos aplicativos e sites que você acessa, tudo ao seu redor é criado e gerenciado com o auxílio de sistemas automáticos que alguém desenvolveu. Esse alguém pode ser (e espero que em breve seja) você! Mas como amplamente mencionado por profissionais da tecnologia, programar é uma arte muito mais ampla do que se pode imaginar.

Maria Klawe é presidente da organização sem fins lucrativos "Math for America" (Matemática para os Estados Unidos) que, apesar do nome, oferece capacitação para professores de todas as disciplinas STEM (ciência, tecnologia, engenharia e matemática). Além de matemática, Maria também é pintora. Ela vê a programação, assim como a pintura, como uma expressão de criatividade que deve ser incentivada desde a juventude.

Mariéme Jamme lidera a iniciativa iamtheCODE ("eu sou o código") com o objetivo de impactar 1 milhão de jovens mulheres e meninas até 2030 através do ensino da programação. Ela destaca a importância de desenvolver o pensamento computacional. Programar não é só escrever código, mas resolver problemas do mundo real através da tecnologia. Segundo Mariéme, os desafios que enfrentamos ao aprender programação são capazes de transformar a maneira que enxergamos o mundo.

Existem concepções equivocadas sobre o ensino da programação, limitando seus objetivos a "aprender a escrever código" ou "aprender uma linguagem de programação". Programar é muito mais do que isso. Envolve não apenas a escrita de código, mas antes disso a formulação de problemas do mundo real como problemas computacionais, o pensamento criativo para propor soluções e a expressão clara dessas soluções através de algoritmos. Transformar uma solução em código executável pelo computador é apenas a etapa final desse processo.

Pensamento Computacional

Portanto nesse curso você vai passar a maior parte do tempo exercitando o pensamento computacional, ou seja, modelando problemas do mundo real como problemas que podem ser resolvidos passo-a-passo por um computador. Para que esse objetivo fique mais claro, vamos usar como exemplo um sistema de streaming, seja Netflix, Spotify, GloboPlay, ou qualquer outro que você costume usar. Todos têm o objetivo de personalizar as recomendações. Cada usuário vê diferentes conteúdos em destaque dependendo do que se costuma consumir na plataforma. Mas como eles transformam algo tão abstrato como as preferências de uma pessoa por filmes, séries ou música em um problema computacional?

Para alcançar o objetivo final, o código executável, precisamos construir um **modelo** do nosso problema, ou seja, uma simplificação do mundo real que torne o problema mais fácil de lidar. Em termos computacionais precisamos definir os **dados** e as **operações**, organizados da seguinte maneira:

Os dados de entrada se referem às informações que sabemos de antemão, o estado inicial do nosso problema. Um sistema de streaming como a Netflix coleta dados de quais filmes o usuário assistiu, se assistiu até o final, se clicou no botão de curtir, além de já possuir inúmeras características de cada filme, como gênero, elenco, direção, entre outros. Em seguida são definidos os dados de saída, ou seja, as informações que se deseja obter. No caso da Netflix, a saída desejada é o conjunto de filmes com características mais similares aos mais assistidos e mais curtidos pelo usuário. A partir dessas informações, o que sabemos e o que buscamos, são definidas regras lógicas e o passo a passo de operações necessário, o algoritmo.

Note que, no problema acima, a tarefa abstrata "gostos do usuário" é substituída por algo concreto e calculável, a "similaridade entre filmes assistidos e não assistidos". Essa é a competência da **abstração** associada ao pensamento computacional. Conheça as 4 competências do pensamento computacional (decomposição, reconhecimento de padrões, desenho de algoritmos e abstração) nesse vídeo com Camila Achutti.

Algoritmo vs Código

Em breve iremos explicar em detalhes tanto o algoritmo quanto o código, mas por hora vale entender que eles são coisas distintas. Enquanto o algoritmo é um esboço conceitual e abstrato da solução, que pode ser formalizado através de descrições textuais ou diagramas, o código remete diretamente a instruções escritas em alguma linguagem de programação que um computador consegue interpretar e executar.

Essa é uma diferença importante quando falamos que nesse curso você vai aprender a resolver problemas, e expressá-los através de código. O código nada mais é que um texto que atende regras gramaticais de alguma linguagem de programação que nos permite expressar algoritmos já idealizados. É quando transformamos uma solução conceitual em um conjunto de instruções bem definidas para o computador. No contexto deste curso, a ferramenta escolhida para esse último estágio será o **Python**.

A parte boa é que ao final deste curso você não ficará preso ao Python. As estruturas de dados e de controle que aprenderemos são compartilhadas por inúmeras outras linguagens, conhecidas como C-like (parecidas com C). A linguagem C é a principal representante de uma família de linguagens como Java, C#, C++, entre muitas outras, incluindo Python, onde o conhecimento deste curso pode ser facilmente aproveitado. Vale ler esse texto do Brian MacDonald da O'reilly, uma das principais editoras de livros de computação, elaborando um pouco mais sobre o quão generalizável é esse conhecimento (lembre-se que você pode traduzir a página para o português no seu navegador).

Mitos e Verdades

Para encerrar essa aula quero desmistificar duas falsas crenças sobre aprender a programar e te contar os reais desafios que você vai encarar.

(FALSO) **Precisa saber muita matemática para programar**. Isso depende muito dos problemas que você vai resolver. Claro que existem áreas como desenvolvimento de jogos, robótica ou ciência de dados, cujas soluções dependem de conhecimento em matemática, física, álgebra, estatística, etc. Já outras áreas, como desenvolvimento web, vão exigir muito menos dessas disciplinas. Além do mais, é cada vez mais comum que as linguagens de programação tragam abstrações prontas e otimizadas de inúmeras funcionalidades, principalmente no campo das exatas, facilitando bastante o seu uso. Raciocínio lógico para fazer o melhor uso dessas abstrações é muito mais relevante.

(FALSO) **Precisa ser fluente em inglês.** Hoje em dia existe muito (muito mesmo) material em português pra aprender! E apesar das linguagens de programação usarem muitas palavras em inglês, não se preocupe, você está literalmente aprendendo uma nova linguagem e nós vamos te ensinar o significado de tudo. Mesmo que as linguagens usassem português, você ainda teria que entender como o computador interpreta aquelas palavrinhas na hora de executar.

(VERDADE) **Só aprende quem pratica muito.** Com o tempo e experiência, todos os problemas serão uma variação de algum problema que você já viu e resolveu no passado. A competência **reconhecimento de padrões** do pensamento computacional envolve reconhecer a similaridade entre problemas, reaproveitando da melhor maneira soluções já existentes. Por isso não se assuste com a quantidade de exercícios e não subestime exercícios fáceis, eles são pensados para reforçar conceitos fundamentais. Mas também não se intimide com exercícios difíceis. Às vezes você vai olhar fixamente pra tela por horas antes de escrever a primeira linha de código. Faz parte :)

(VERDADE) **O conteúdo é cumulativo.** Muita gente desiste de aprender a programar afirmando que "não consegue avançar". Aprender bem as bases é extremamente importante, já que assuntos avançados são construídos em cima de estruturas básicas. Por isso não deixe nenhuma dúvida passar sem resposta e principalmente não tenha pressa de avançar. Uma boa fundamentação vai te levar muito mais longe.

Agora que você já sabe o que esperar dessa jornada, vamos dar os primeiros passos nesse curso que é a base das mais diferentes áreas do desenvolvimento tecnológico! Espero tornar esse processo tranquilo e quem sabe até divertido **para você!**

O que é um algoritmo?

Vamos partir de uma definição que você vai encontrar por aí na Internet e decodificar o significado das palavras-chave geralmente utilizadas:

"Sequência finita de instruções bem definidas e não ambíguas para realizar uma tarefa"

A primeira parte é fácil, **sequência finita de instruções.** Pode ser um manual de montagem, uma receita culinária, instruções de uso de um medicamento, ou qualquer outra sequência de instruções organizadas em um passo-a-passo que outras pessoas consigam seguir. Pense por exemplo no problema de aprender a tabuada de multiplicação. É comum que estudantes decorem a tabuada e, quando precisam usar aquele conhecimento, respondam de memória sem seguir um raciocínio passo-a-passo. Essa solução **não é algorítmica.**

Por outro lado, eu aprendi na escola um truque muito bacana para a tabuada de multiplicação do 9. Multiplicar o 9 por qualquer número n entre 1 e 10 pode ser descrito como:

1. Subtraia 1 de n, ou seja, n-1. Esse será o primeiro dígito
2. Subtraia n de 10, ou seja, 10-n. Esse será o segundo dígito.

9x1 =	09
9x2=	18
9x3=	27
9x4=	36
9x5=	45
9x6=	54
9x7=	63
9x8=	72
9x9=	81
9x10=	90

Veja os resultados um por um na imagem ao lado. E note como conseguimos enumerar passos sequenciais que qualquer pessoa (capaz de realizar subtrações) consegue seguir e garantidamente chegar na mesma resposta. Isso sim **é um algoritmo.**

Vamos entender agora o que significa sem bem definido e não ambíguo. Você já tentou seguir uma receita de cozinha da Internet com instruções abstratas como "deixe no fogo por mais ou menos 40 minutos". Assumindo que, assim como eu, você não vai muito bem na cozinha (espero que não seja o caso), essa instrução não deixa claro o que devo levar em consideração para decidir se deixo mais de 40 minutos ou menos. Ela é mal definida para mim, mas pode ser suficiente para profissionais da culinária. Portanto, ser bem definido significa ter um nível de detalhe suficiente para quem vai ler e executar as instruções. No caso da programação de software, nosso público-alvo é o computador, um dos bichos mais burros e literais com quem você vai conversar.

Vale assistir também esse vídeo onde um pai pede que seu filho escreva o algoritmo para fazer um sanduíche de pasta de amendoim com geléia. Ele tenta simular como um computador responderia, executando literalmente e somente o que está descrito no papel. O computador não vai preencher as lacunas para você, ou executar algo que parece óbvio mas não está descrito. Por isso precisamos pensar com calma todos os aspectos do problema.

Essa experiência de pai e filho também nos ajuda a entender o passo-a-passo da construção de algoritmos. Assim como o filho tenta várias vezes, e escreve algoritmos cada vez melhores, a escrita de algoritmos para o computador também é um processo de refinamento. Em geral vamos esboçar soluções de maneira genérica, elencando os sub-problemas daquela tarefa, para só então definir ações específicas. Já que estamos falando de culinária, vamos fazer isso com uma receita de bolo. Podemos começar com as seguintes instruções.

Algoritmo receita de bolo;
1- Separe os ingredientes do bolo;
2- Misture os ingredientes separados;
3- Ponha para assar;
4– Retire do fogo.
Fim do algoritmo

Em seguida resolvemos, um a um, cada sub-problema elencado, aumentando o nível de detalhe da solução. Algo parecido com:

Algoritmo receita de bolo;
Separe os ingredientes do bolo
1- Separe 3 ovos, 250ml de leite, 50g de manteiga, 300g de açúcar, 300g de farinha de trigo;
Misture os ingredientes separados{
2- Adicione a manteiga, o açúcar e os ovos;
3- **Enquanto** a mistura não estiver homogênea:
4- Misture;
5- Adicione a farinha e o leite;
6- **Enquanto** a mistura não estiver homogênea:
7- Misture;
Ponha para assar
8- Ligue o forno a 180 graus;
9- Unte a forma com manteiga;
10- Ponha a massa na forma untada;
11- Ponha a forma com a massa no forno;
Retire do fogo
12- Espere 30 minutos;
13- **Enquanto** a forma estiver no forno:
14- Abra o forno e enfie um palito no bolo;
15- Se o palito sair limpo, então:
16- retire a forma do forno;
17- **Senão:**
18- espere 5 minutos;
Fim do algoritmo

Aqui estamos assumindo que a pessoa cozinhando sabe o que é uma mistura homogênea, sabe untar uma forma, entre outros. Então não precisamos refinar infinitamente, basta saber qual nível de conhecimento prévio deve ser esperado. No caso do computador, as linguagens de programação trazem uma série de funcionalidades pré-prontas que podemos usar para compor nossas soluções.

Algoritmos para o Computador

Vamos conhecer as 3 maneiras de representar algoritmos: dissertativa, fluxograma e pseudocódigo. Independente da representação, escrever algoritmos começa identificando os mesmos elementos que comentamos no capítulo 1.1 ao falar de pensamento computacional: a **entrada**, as operações de **processamento** e a **saída** esperada. Esses 3 elementos devem ser convertidos em instruções para o computador. Vamos fazer isso para o problema simples de somar dois números (em breve faremos uma calculadora completa!).

- **Entrada: os números que se deseja somar;**
- **Processamento: operação de soma aplicada sobre os números de entrada;**
- **Saída: resultado da soma;**

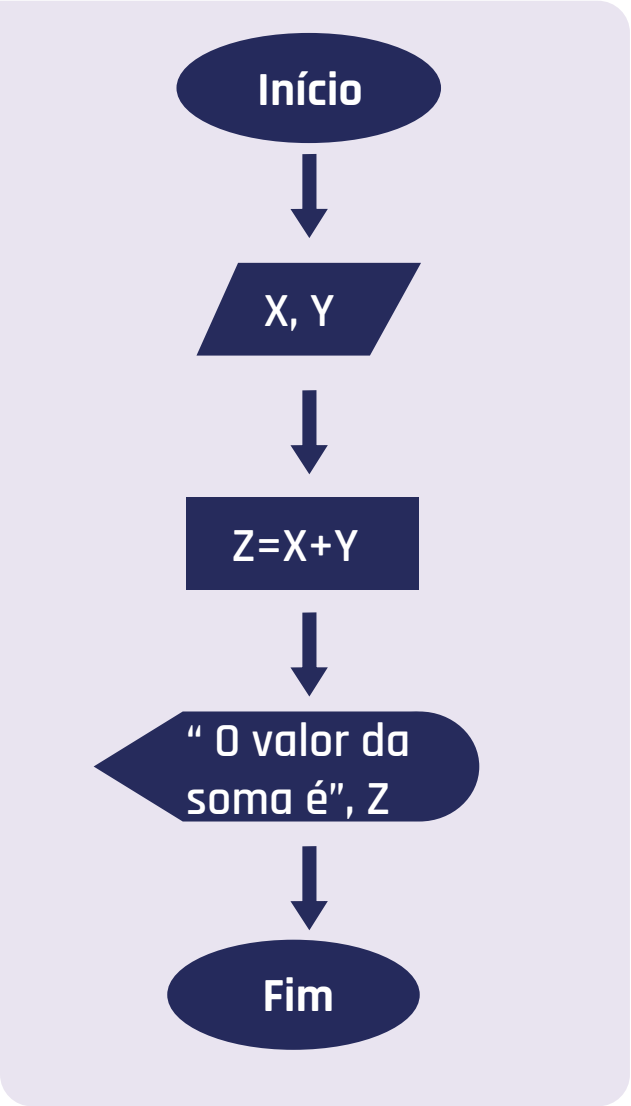
1 - **Leia** do teclado e **armazene** os dois números que se deseja somar; #Entrada
2 - Aplique a **operação** de soma sobre os dois números recebidos e **armazene** o resultado; #Processamento
3 - **Apresente** na tela o resultado da soma e finalize o programa; #Saída

Note que usamos frases como “leia do teclado” para indicar a origem dos dados, e “apresente na tela” para indicar a forma da saída. Ao longo do curso conheceremos diferentes maneiras que o computador e as linguagens de programação nos permitem carregar dados no nosso programa e as ações comumente aplicadas aos resultados de algoritmos.

Já o fluxograma é uma representação gráfica do algoritmo, usando símbolos pré-definidos para representar instruções. O mais importante aspecto dos fluxogramas são as setas que conectam as instruções, definindo o fluxo do programa, ou seja, a ordem de execução das instruções. Aprenderemos uma série de desvios de fluxos causados por estruturas de controle, chamadas de funções, entre outros. Manter o controle do fluxo é essencial para compreender a execução do algoritmo. A seguir estão os possíveis símbolos utilizados em um fluxograma.

Para resolver a soma de dois números, mais uma vez definimos entrada, processamento e saída. Note que o símbolo utilizado já separa os três elementos visualmente. Opcionalmente podemos manter os verbos “leia” e “imprima” para as instruções de entrada e saída.

O fluxograma está mais próximo do código fonte, já que usamos o conceito de variáveis para nomear os operandos lidos e processados (X, Y, Z), além de operadores (como a soma) e símbolos associados a instruções (como o sinal de igualdade para armazenar resultados).



Por fim, o pseudocódigo é uma forma intermediária entre uma representação dissertativa e o código de programação real. Ele utiliza uma linguagem simplificada e próxima à linguagem natural, incorporando algumas convenções de programação (operadores aritméticos, armazenamento em variável, etc.), mas sem estar vinculado a uma linguagem específica.

```
1- Início
2- Leia X, Y; #Entrada
3- Z = X + Y; #Processamento
4- Imprima Z; #Saída
5- Fim
```

Em resumo, algoritmos são instruções direcionadas para algo ou alguém executar. Para escrevê-los, precisamos conhecer as operações que o nosso público-alvo é capaz de realizar Para que o computador seja capaz de executá-lo ainda falta o próximo passo: transformar algoritmo em código fonte.

Exercício

1) Dadas as regras do truque da tabuada do 9, escreva a solução algorítmica das 3 maneiras que conhecemos: dissertativa, fluxograma e pseudocódigo.

Regras: O resultado da multiplicação do 9 por qualquer valor n é calculado por dígito, da seguinte maneira:
9 x n = n-1 10-n

O resultado final deve multiplicar o primeiro dígito (a dezena) por 10 e somar com o segundo dígito (a unidade)
9 x n = **(**n-1**)** x 10 + **(**10-n**)**

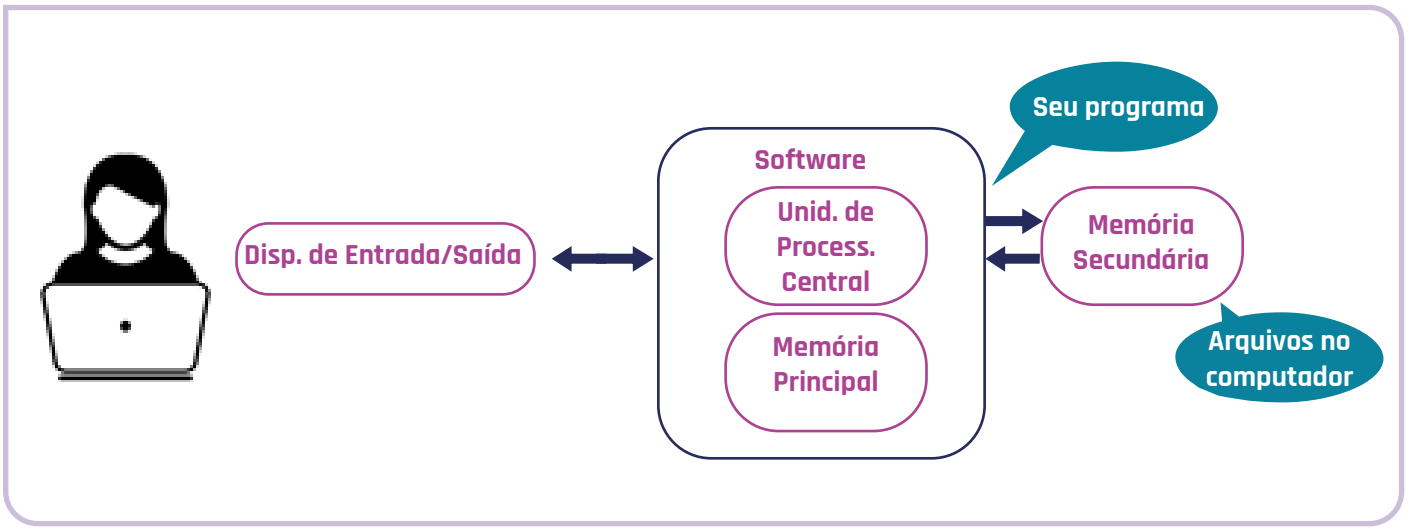
Lembre-se de seguir o passo a passo de entrada (leitura dos dados necessários), processamento (operações listadas acima) e saída (impressão do resultado final).

Instruções para submissão: Na pasta modulo1 do seu repositório no Github coloque os seguintes arquivos:

a)dissertativa.txt: um arquivo de texto com a solução dissertativa.
b)pseudocodigo.txt: um arquivo de texto com a solução em pseudocódigo.
c)fluxograma.jpg: um arquivo de imagem com o fluxograma, podendo ser:
i)Uma foto do fluxograma desenhado no papel.
ii)Um desenho digital. Dica: crie uma cópia desse arquivo no Google Drawings e reorganize os elementos do fluxograma.

Do algoritmo ao código fonte

Pra escrever código (e melhores algoritmos) precisamos entender nosso público-alvo, quem vai executar o código, ou seja, o computador. Vamos partir da simplificação a seguir apresentando a relação entre os componentes físicos do computador.



1. **Unidade Central de Processamento (CPU):** Responsável por executar as instruções de um programa. Execução rápida de operações aritméticas e lógicas, além de manipulação de dados.

2. **Memória Principal:** Conhecida como RAM (Memória de Acesso Aleatório - Random Access Memory). Armazena dados temporariamente para o processador acessar rapidamente; volátil e usada durante a execução de programas.

3. **Memória Secundária:** Armazenamento de longo prazo para dados e programas; usada para salvar informações mesmo quando o computador está desligado (exemplos: disco rígido, SSD, cartão de memória).

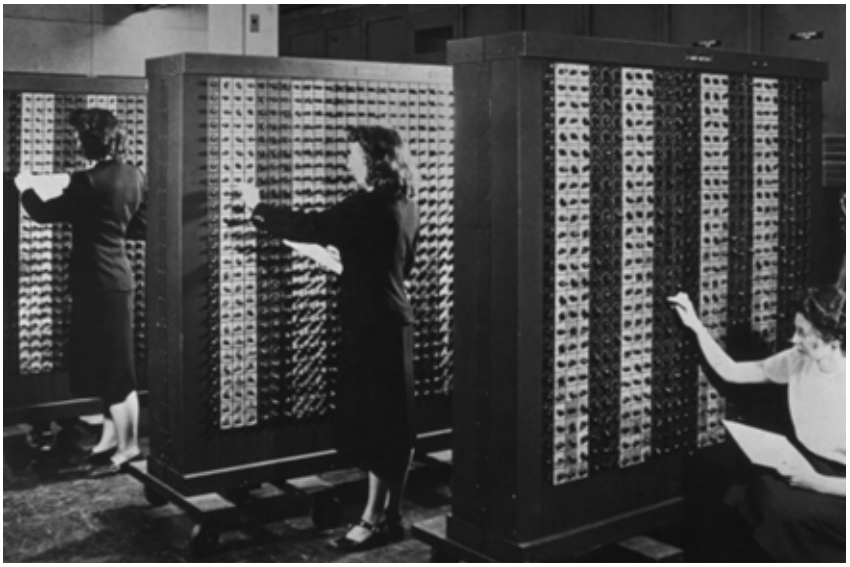
4. **Dispositivos de entrada:** Permitem a entrada de dados no computador; exemplos incluem teclado, mouse, tela sensível ao toque, etc.

5. **Dispositivos de saída:** Exibem ou fornecem resultados do processamento; exemplos incluem monitor, impressora, e alto-falantes.

Falamos anteriormente que um algoritmo é composto por dados de entrada, dados de saída e operações de processamento. As limitações de cada um desses elementos é definida pelos componentes diretamente associados a eles, ou seja, de onde vem as entradas, onde coloco as saídas e quais as operações e restrições existentes. Por exemplo, entradas podem ser provenientes de um dos dispositivos de entrada disponíveis ou de arquivos da memória secundária, e está limitada aos tipos de dados que a CPU é capaz de manipular. Idem para os dados de saída, que podem ser apresentados via dispositivos de saída ou armazenados na memória secundária, além de ser também limitados pela capacidade de manipulação de dados da CPU. Por fim, o processamento se relaciona diretamente com a capacidade física da unidade de processamento, a qual executa operações aritméticas e lógicas.

Linguagens de programação

Linguagens de programação definem regras gramaticais de alto nível, ou seja, legíveis para o ser humano, definindo instruções que podem ser traduzidas em linguagem de máquina. Lembre-se que internamente tudo é binário (zeros e uns) no computador. Antes das linguagens de programação, as operadoras dos primeiros computadores programavam ligando (1) e desligando (0) centenas de lâmpadas para escrever código binário.



Se hoje podemos escrever programas de forma legível, é resultado de uma constante evolução das linguagens e do trabalho de pessoas como a Grace Hopper, dedicando suas carreiras ao desenvolvimento de compiladores ou interpretadores capazes de traduzir linguagens de alto nível para linguagem de máquina. Obrigada Grace!

Linguagem de máquina

```
0001001001000101
0010010011101100
10101101001...
```

Baixo nível

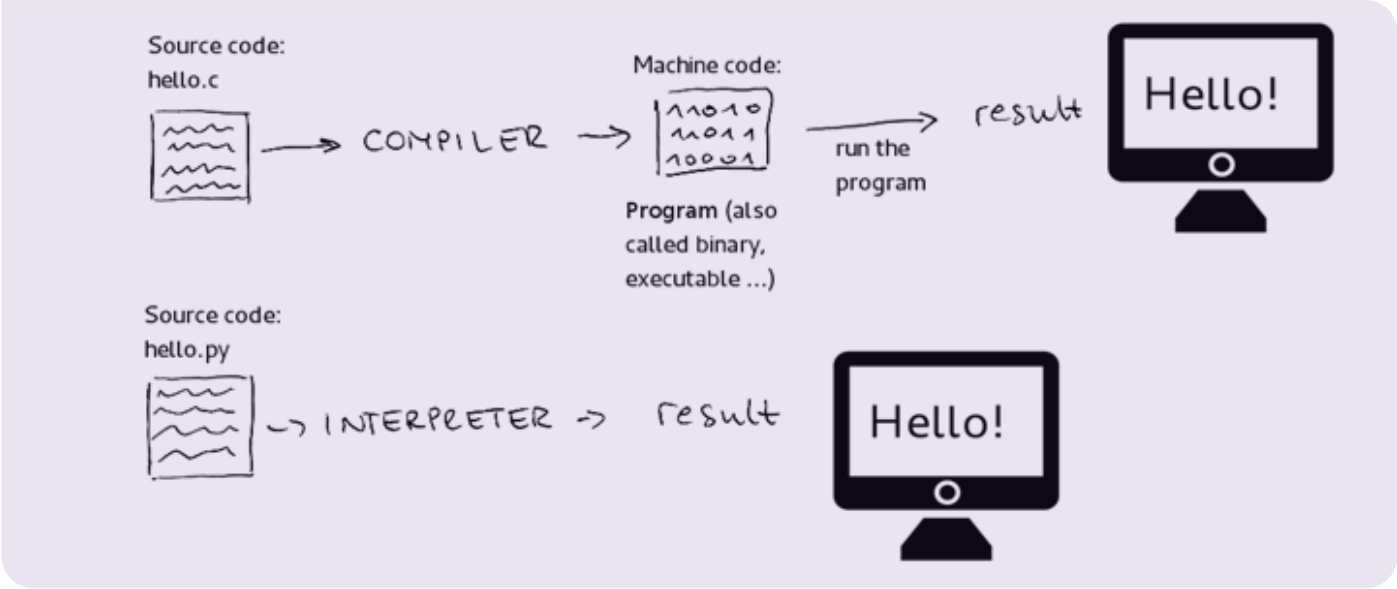
```
LOAD r1,b
LOAD r2,h
MUL r1,r2
DIV r1,#2
RET
```

Alto nível

```
class Triangle {
  ...
  float surface()
    return b*h/2;
  }
```

O código fonte é apenas um arquivo de texto que obedece às regras de uma linguagem de programação. Nesse curso conheceremos as regras gramaticais da linguagem Python, tanto em termos de sintaxe quanto semântica. Em termos de código, sintaxe se refere à organização das palavras em frases, ou melhor, de **tokens** (palavras e símbolos da linguagem) em **instruções**. Já a semântica é o significado dessas instruções, ou o resultado de sua execução pelo processador.

Apesar do código poder ser escrito em qualquer editor de texto, sua execução depende de ferramentas capazes de converter a linguagem de alto nível em linguagem de máquina. Existem duas principais maneiras que isso pode ser feito, através da compilação ou da interpretação. A figura a seguir nos ajuda a entender a diferença. Enquanto a compilação é um processo de tradução que salva o código binário de todo o programa para execução futura, o interpretador lê e executa linha a linha do script, tendo como saída o resultado da execução. Você também pode assistir esse vídeo que explica a diferença entre os dois processos usando receitas culinárias.



Python é uma **linguagem interpretada**, ou seja, mais dinâmica durante o desenvolvimento, já que não é necessário recompilar todo o código a cada pequena mudança. Compilar é um processo demorado, o que virou até piada no meio do desenvolvimento, já que criou-se o hábito de sair pra tomar café ou jogar conversa fora "enquanto o código compila". Por outro lado, é mais rápido executar um programa depois de compilado, que já está em linguagem de máquina e pode ser diretamente entendido pelo computador.

Existem ambientes de programação com as duas funções, editor de texto e interpretador ou compilador de código. São conhecidos como IDE (Integrated Development Environment - ambiente integrado de desenvolvimento). Python tem vários! Nesse curso iremos alternar entre o Visual Studio Code, o Jupyter e o Google Colab. É sobre elas que falaremos na próxima aula.

Ferramentas

Ambientes de desenvolvimento ou, como chamaremos a partir de agora, IDEs são ambientes unificados para escrita, edição, compilação/interpretação e depuração de códigos. Falaremos desse último (depuração) em disciplinas futuras. Quanto à escrita e edição, IDEs têm vantagens em relação a editores de textos genéricos, trazendo funcionalidades como colorir o texto de acordo com o significado de cada token, complemento automático de palavras e em sistemas inteligentes mais recentes até completando funções inteiras.

```python
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier


def make_rf_predictions(X: np.ndarray, y: np.ndarray, X_test: np.ndarray) -> np.ndarray:
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X, y)


def ds_logic():
    train_data = pd.read_csv("/kaggle/input/titanic/train.csv")
    test_data = pd.read_csv("/kaggle/input/titanic/test.csv"
    y = train_data["Survived"]

    features = ["Pclass", "Sex", "SibSp", "Parch"]
    X = pd.get_dummies(train_data[features])
    X_test = pd.get_dummies(test_data[features])
    predictions = make_rf_predictions(X.values, y.values, X_test.values)
```

A IDE também está associada ao interpretador da linguagem que dá suporte, ou compilador dependendo da linguagem, sendo responsável por dar um retorno visual facilitado sobre a execução do código. Códigos sem nenhum erro de sintaxe ou semântica são executados dentro da própria IDE. Por outro lado, e esse é o aspecto mais importante, a ferramenta apresenta detecção de erros tanto em tempo real (antes de ser executado), quando após uma tentativa de execução. Falaremos muito sobre mensagens de erro e como elas nos ajudam a diagnosticar e corrigir nossos códigos. Mensagens de erro serão suas melhores amigas, não só nessa etapa mas em toda a sua carreira.

```
In [4]: print("Ola mundo"

        Cell In[4], line 1
          print("Ola mundo"
                            ^
SyntaxError: unexpected EOF while parsing
```

Importante: As informações desta aula podem ser melhor absorvidas a partir das vídeo-aulas. O entendimento das ferramentas é mais prático do que teórico.

Python Interativo

O primeiro ambiente que conheceremos é o Python Interativo. Antes de falarmos dele, verifique se já está tudo instalado.

·Abra um terminal, digite o comando a seguir e aperte enter

>>> python3 --version

·Verifique se o texto resposta é a versão Python 3.8, ou superior

Caso não esteja instalado, bora instalar?

·No Ubuntu 18.04 ou superior, abra o terminal e entre os seguintes comandos:

>>> sudo apt-get update

>>> sudo apt-get install python3.8 python3-pip

·No Windows 10, baixe o instalador do Python 3.11 no site do Python:

·Na seção em Stable Releases encontre o Download Windows Installer (64-bit) da versão mais recente do Python;

·Execute o arquivo baixado e siga as instruções do instalador.

Agora é só fazer o primeiro teste de novo para garantir que a instalação aconteceu com sucesso. Tudo certo? Vamos lá então. O ambiente interativo do Python, acessível através do terminal, oferece uma plataforma dinâmica e eficiente para explorar, testar e executar código Python de forma interativa. Ao digitar python3 no terminal, entramos no interpretador interativo, onde podemos inserir linhas de código Python e obter imediatamente os resultados.

```
$ python3
Python 3.9.6 (default, Oct 18 2022, 12:41:40)
[Clang 14.0.0 (clang-1400.0.29.202)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

O símbolo >>> na última linha indica que o ambiente está aguardando a entrada de um comando Python. Ainda não conhecemos comandos, mas podemos testar como uma calculadora. Cada operação é lida pelo interpretador, transformada em linguagem de máquina, executada pelos componentes físicos responsáveis (cpu, memórias, dispositivos) e o resultado é apresentado na linha seguinte. Ao final de cada execução o símbolo >>> reparece aguardando novo comando.

```
>>> 12 - 4
8
>>> 3 * 7
21
>>>
```

```
>>> 2 +
  File "<stdin>", line 1
    2 +
      ^
SyntaxError: invalid syntax
>>> exit()
$
```

Em caso de erros de sintaxe ou semântica na instrução dada, o erro apontado pelo interpretador é apresentado na linha seguinte à instrução incorreta. Para sair, digite **exit()**.

Assim como no terminal do Linux, as setas para cima e para baixo te permitem navegar no histórico de comandos digitados. Porém, ao encerrar o ambiente perdemos acesso a todas as instruções executadas na interação, afinal elas não foram salvas em arquivo em nenhum momento. Por isso o ambiente é muito utilizado para testes curtos e rápidos de pequenos trechos de código. É uma abordagem ágil para experimentação, permitindo testar ideias, verificar rapidamente a sintaxe, e interagir diretamente com as funcionalidades da linguagem.

A alternativa para soluções mais longas e complexas é escrever arquivos de código, no caso do Python chamados de scripts. Por convenção, a extensão .py é adotada para que IDEs reconheçam um dado arquivo como um **script Python.** Como falamos anteriormente, scripts podem ser escritos em qualquer editor de texto, mas a sua execu-

ção obrigatoriamente precisa do interpretador. Este pode ser acionado pelo terminal do sistema operacional.

Da mesma forma que iniciamos o ambiente interativo invocando o comando python3 no terminal, podemos executar scripts salvos em arquivos com uma pequena modificação. Basta adicionar ao comando o nome do arquivo que contém o script, separado por um espaço em branco. No exemplo a seguir, o arquivo operacoes.py é executado no terminal ao invocar o comando python3 operacoes.py.

operacoes.py	**terminal**
print(12-4) print(3*7)	$ python3 operacoes.py 8 21 $

Diferente do ambiente interativo, onde o resultado de cada expressão é apresentado após a sua execução, no script temos que invocar explicitamente a apresentação de resultados. Para isso você precisa conhecer o seu primeiro comando: print(), que recebe dentro dos parênteses uma expressão e apresenta no terminal o resultado da execução. Experimente criar um arquivo de texto com o nome "ola.py" contendo o código a seguir e o execute no terminal.

```
print("Olá mundo!")
```

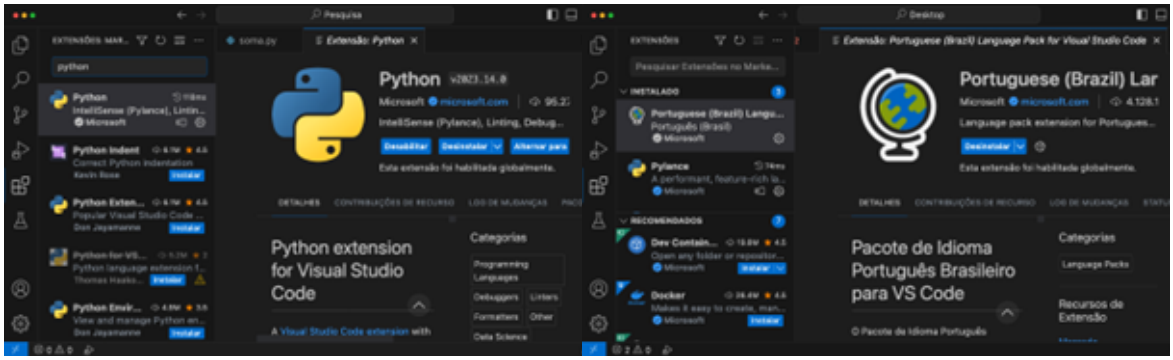
Visual Studio Code

O Visual Studio Code é uma IDE leve e poderosa desenvolvida pela Microsoft, que dá suporte a muitas linguagens de programação. Ela se destaca como uma das IDEs mais utilizadas graças a sua extensibilidade (inúmeras extensões que complementam suas capacidades) e sua facilidade de integração com outras ferramentas, como as de controles de versão.

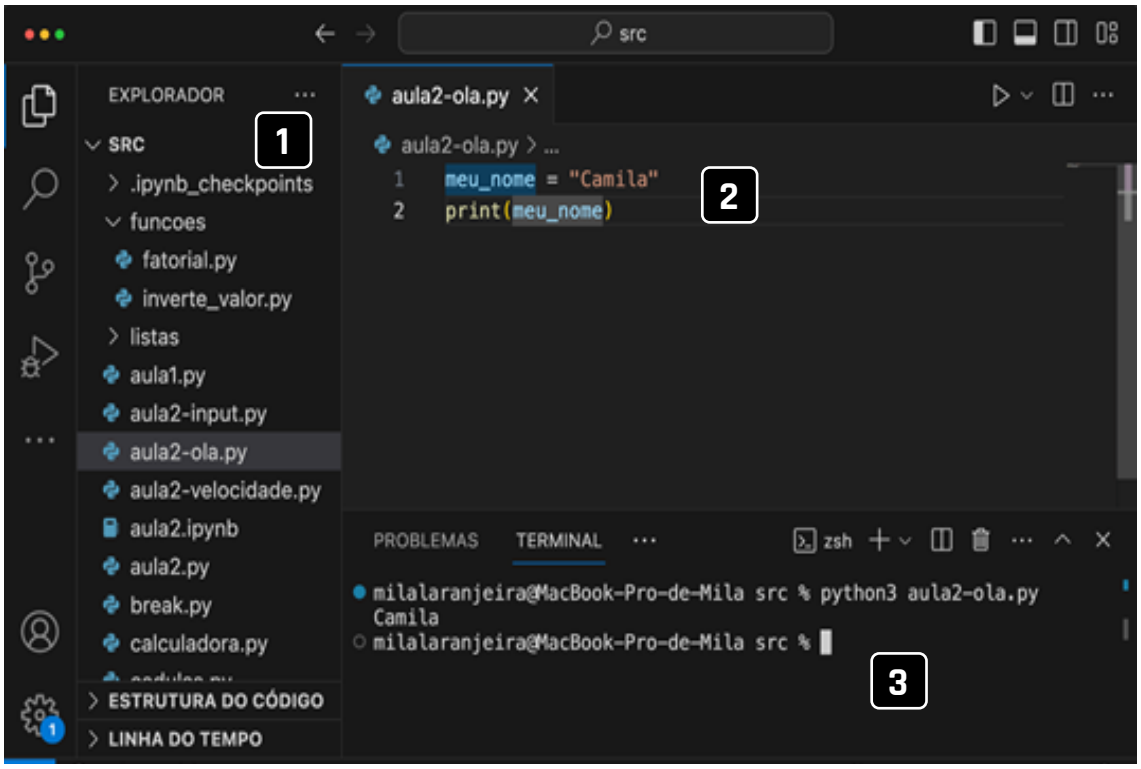
Antes de tudo, verifique se o VSCode (como chamam os íntimos) está instalado procurando o software entre os programas do seu computador. Se não tiver, é só ir no site e baixar o instalador para o seu sistema operacional. Basta clicar no grande botão azul

com o nome do seu sistema operacional (Windows 10 ou .deb para Ubuntu), executar o arquivo que será baixado e seguir as instruções de instalação.

O VSCode é baseado em extensões. Para começar a trabalhar precisamos de duas: a extensão Python com o interpretador da linguagem e a Portuguese (Brazil) Language para que a ferramenta alterne para o nosso idioma.



IDEs como o VSCode são compostas de três janelas principais, **(1) o explorador de arquivos, (2) editor de código e (3) terminal.**



O explorador permite que você visualize e navegue nos arquivos salvos no seu computador e de maneira simples abra um script existente ou inicie um novo script, o qual será aberto para edição no editor de código. Aos poucos conheceremos as vantagens do editor, mas de primeira já conseguimos ver as diferentes cores no script, indicando que cada palavra ou símbolo representa algo específico para o interpretador. Teste também escrever uma palavra, qualquer palavra, e veja o editor

sugerindo complementos automaticamente.

Diferente do Python Interativo, os resultados das operações precisam ser armazenados em variáveis e impressões de resultados precisam ser explicitamente chamadas através do comando print, coisas que só aprenderemos mais pra frente. Portanto não se apresse, em breve faremos muitos scripts (muito mesmo!).

Jupyter Notebook

Meu favorito pessoal, mistura os dois mundos: escrevemos scripts, mas de maneira interativa (em pequenos blocos/células). Além disso, permite a criação de notebooks, como seus arquivos são chamados, que misturam texto formatado e blocos executáveis de código, ideal para fins educacionais ou exploração de dados. A extensão de seus arquivos é .ipynb (interactive Python notebook - notebook Python interativo).

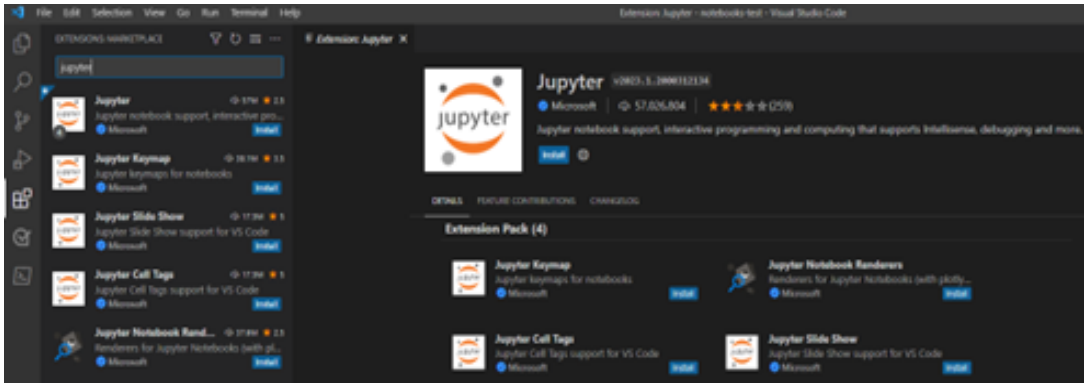
Primeiro verifique se ele está instalado. Abra o terminal e digite:

```
>>> jupyter notebook
```

Se tudo estiver certo, ele vai abrir no navegador padrão do seu computador, já que se trata de uma aplicação web. Mas mesmo assim, ele executa localmente e te dará acesso aos arquivos salvos no seu computador. Se não tiver, é só instalar. O Jupyter faz parte do gerenciamento de pacotes padrão do Python, o pip. É através dele que instalaremos as principais bibliotecas do Python. É só abrir o terminal e digitar:

```
>>> pip install notebook
```

É possível abrir notebooks Jupyter no VSCode, basta instalar a extensão de mesmo nome.



Para iniciar um notebook:

•No VSCode basta criar um arquivo de qualquer nome com a extensão .ipynb e ele será reconhecido como um notebook interativo.

•No Jupyter clique no botão **New** no canto superior direito e escolher a opção Python 3.

Ao abrir o notebook você verá apenas uma célula vazia, que por padrão está configurada para escrita de código Python. Existem dois tipos principais de células: código e markdown/texto. Células de código são usadas para escrever e executar código, enquanto células markdown permitem escrever textos formatados. Não falaremos tanto da formatação de texto, mas você pode consultar aqui as principais formatações disponíveis. Usamos essas codificações de texto em células Markdown, como apresentado a seguir.

```
1 # Módulo 6 - Listas
2
3
4 ## 6.3 Fatiamento
5
6 Dada uma lista qualquer, encontre o intervalo que possui a maior quantidade de números negativos e delete ele
  da lista com o operador del
```

As células de código permitem uma execução interativa, o que significa que você pode executar porções específicas de código em vez de executar um script inteiro. Nesse sentido, o Jupyter se aproxima do Python Interativo, já que ao executar uma célula, vemos imediatamente o seu resultado, e quaisquer mudanças de estado persistem enquanto o notebook estiver ativo. Isso vai fazer mais sentido quando conhecermos variáveis, fluxo de código, e outros fatores que interferem nas mudanças de estado.

Vamos conhecer melhor cada uma dessas ferramentas à medida que as usar-

mos. Na imagem a seguir temos um exemplo de notebook jupyter composto respectivamente por uma célula de texto e uma célula de código, ambas executadas. Enquanto a célula de texto omite os códigos de formatação, mantendo apenas os textos formatados, a célula de código apresenta abaixo dela o resultado da execução.

Módulo 6 - Listas

6.3 Fatiamento

Dada uma lista qualquer, encontre o intervalo que possui a maior quantidade de números negativos e delete ele da lista com o operador del

```
1  # mesma lógica de encontrar o maior número e dizer a sua posição,
2  # precisaríamos guardar o maior valor, e sua posição.
3
4  nums = [19, 22, -10, 4, -2, -3, 5, 6, -17, -4, -1, 67, 98, -23, -6]
5
6  idx_maior, val_maior = 0, 0
7
8  idx_atual = 0 # iniciando iteração nos índices
9  for val in nums: # iterando nos valores
10     print(idx_atual, val, end = ' ')
11     if val > val_maior:
12         val_maior = val # registrando maior valor (até agora)
13         idx_maior = idx_atual # registrando índice do maior valor (até agora)
14         print('maior até agora')
15
16     idx_atual += 1 # iterando nos índices
17
18 print('\nMaior ao final do laço:', idx_maior, val_maior)
```

```
0 19 maior até agora
1 22 maior até agora
2 -10 3 4 4 -2 5 -3 6 5 7 6 8 -17 9 -4 10 -1 11 67 maior até agora
12 98 maior até agora
13 -23 14 -6
Maior ao final do laço: 12 98
```

Google Colab

Por fim, o Google Colab segue a mesma lógica de células do Jupyter Notebook. Seu diferencial, que também é sua principal vantagem, é ser um serviço de nuvem gratuito, ou seja, uma ferramenta online sem necessidade de instalação. Basta estar logado em uma conta Google e abrir o site da ferramenta para usufruir dos seus recursos. Ao abrir um novo notebook, você estará acessando a infraestrutura da Google, e a execução acontece nos servidores da empresa (e não localmente em seu computador como faz o Jupyter).

A seguir temos uma imagem exemplo de código no Colab, com uma célula de texto e uma de código, ambas executadas, assim como mostramos anteriormente para o Jupyter. Note também que temos recursos não essenciais, mas muito divertidos, como a possibilidade de adicionar cães e gatos caminhando no topo do script. Para mais detalhes sobre o Colab, que vai te ajudar a consolidar seus conhecimentos sobre notebooks interativos, confira esse artigo publicado pela Alura.

listas.ipynb

Arquivo Editar Ver Inserir Ambiente de execução Ferramentas Ajuda Última edição em 27 de outubro

+ Código + Texto

Exercícios - Lista

Faça um programa que gere aleatoriamente 20 valores inteiros entre -100 e 100. Em seguida imprima na ordem estabelecida:

- A lista ordenada, sem modificar a lista original
- A lista original
- O índice do maior valor da lista
- O índice do menor valor da lista

```
[ ]  1 import random
     2
     3 # Construção da lista de valores aleatórios
     4 aleatorios = []
     5 for i in range(20):
     6   valor = random.randint(-100, 100)
     7   aleatorios.append(valor)
     8
     9 ### aleatorios.sort() modifica a lista original
    10 print(sorted(aleatorios))
    11 print(aleatorios)
    12 print("O maior valor está no índice: ",
    13       aleatorios.index(max(aleatorios)) )
    14
    15 print("O menor valor está no índice: ",
    16       aleatorios.index(min(aleatorios)))
```

```
[-87, -76, -75, -66, -65, -48, -48, -45, -40, -34, -21, -11, 19, 35, 40, 43, 63, 71, 78, 100]
[100, -76, 43, -66, 35, 71, -48, -65, -21, 40, -11, 78, -75, 19, -45, -87, 63, -40, -48, -34]
O maior valor está no índice:  0
O menor valor está no índice:  15
```

Exercício

Copie e cole os seguintes códigos em todas as ferramentas que conhecemos, ou seja, Python Interativo (terminal), Visual Studio Code, Jupyter e Google Colab. No terminal você deve copiar uma instrução por vez interativamente (melhor ainda se você digitar letra por letra 😀)

Ex	Entrada	Saída esperada
01	`print("Meu primeiro código")`	`Meu primeiro código`

02	`x = 10`
	`y = 5`
	`print(x+1)`
	`print(y-2)`
	`print(y*x)`
	`print(x/y)`
		`11`
		`3`
		`50`
		`2.0`

03	`print("Ola)`	`print("Ola)`
		`^`
		`SyntaxError: unterminated string literal (detected at line 1)`

Instruções para submissão: Na pasta modulo1 do seu repositório no Github coloque arquivos de imagem com as **capturas de tela** dos três exercícios acima, executados em cada ferramenta. A captura deve mostrar ambos o script e a saída da execução. Para as plataformas de script interativo (Jupyter e Colab) execute todos os códigos **no mesmo notebook**, em células distintas.
Os arquivos devem ter os seguintes nomes:

a) ex01_interativo.jpg
b) ex02_interativo.jpg
c) ex03_interativo.jpg
d) ex01_vscode.jpg
e) ex02_vscode.jpg
f) ex03_vscode.jpg
g) jupyter.jpg
h) colab.jpg

Fundamentos

O que é Variável

Variáveis são "etiquetas" de valores, ou seja, nomes que associamos a valores suportados pela linguagem ou resultados de expressões. Permite referenciar ou manipular valores ao longo do programa através de um nome escolhido por nós. Usamos o operador de atribuição, descrito a seguir, para associar uma variável ao valor que se deseja armazenar.

Operador de atribuição (=)

Associa um valor a um nome de variável. No lado esquerdo sempre teremos o nome da variável que irá referenciar o resultado da expressão à direita do operador de atribuição.

Forma geral: <variável> = <expressão>

Exemplo prático:

```
x = 10
y = 2 + 3
z = x * y
```

O exemplo prático traz 3 atribuições, criando a variável de nome x e associando ao valor 10, a variável de nome y recebendo o resultado da expressão 2 + 3, ou seja, o valor 5. Por fim, criamos a variável de nome z e armazenamos o resultado da multiplicação de duas variáveis pré-existentes, x * y, que representam respectivamente os valores 10 e 5.

Variáveis são compostas por **nome** e **tipo**. Conheceremos os tipos do Python, mas por hora considere por exemplo que os números 10 e 2.5 pertencem a diferentes conjuntos numéricos, respectivamente o conjunto dos inteiros e o conjunto dos reais; enquanto a palavra "olá" pertence a um terceiro conjunto, o textual. Nas linguagens de programação é similar, cada tipo tem características e comportamentos específicos que a linguagem dá suporte.

No Python, diferente de outras linguagens, não precisamos definir explicitamente o tipo das variáveis, que são definidos dinamicamente a partir do dado ali armazenado. Python possui o comando type(), que recebe dentro dos parênteses um valor (expressão ou variável) e retorna o nome do seu tipo. Por exemplo considere as seguintes instruções:

```
>>> x = 10
>>> type(x)
<class 'int'>
>>> x = "olá"
>>> type(x)
<class 'str'>
```

Na primeira linha criamos uma nova variável, chamada x, associada ao valor 10. Na segunda linha consultamos o seu tipo e temos como retorno o nome <class 'int'>, indicando um dado do tipo inteiro. Na terceira instrução substituímos o valor da variável x já existente, colocando agora o texto "olá", e isso altera dinamicamente o tipo da variável e portanto o retorno da consulta type().

Nomes Variáveis

Existem regras para dar nomes às variáveis. Mas antes de falar das obrigatórias, ou seja, as que o interpretador do Python vai verificar antes de executar, vamos falar da mais importante: **escolha bons nomes!** Nomear variáveis de forma intuitiva facilita a leitura do código, não só por outras pessoas, mas você mesmo no futuro. Considere os dois fluxos de execução abaixo:

```
>>> a = 10
>>> b = a * 2
>>> b
20
```

```
>>> raio = 10
>>> diametro = raio * 2
>>> diametro
20
```

Na execução da esquerda não conseguimos inferir pela leitura o que cada variável representa no mundo real. Lembre-se que nossos códigos vão automatizar tarefas reais, que têm significado. Na execução da direita podemos interpretar, sem nenhum texto auxiliar, que se trata do cálculo do diâmetro de um círculo a partir de seu raio. Muito mais fácil, certo?

Quanto às regras obrigatórias, temos 4 pontos essenciais que se desobedecidos vão causar erros de execução (que as IDE nos apontam e precisamos aprender a ler).

- Deve iniciar com uma letra ou underscore (_)
  - Não inicia com números!
- Pode conter letras, números e underscores
  - Não pode conter símbolos além do underscore!
- Case Sensitive (diferencia maiúsculas de minúsculas)

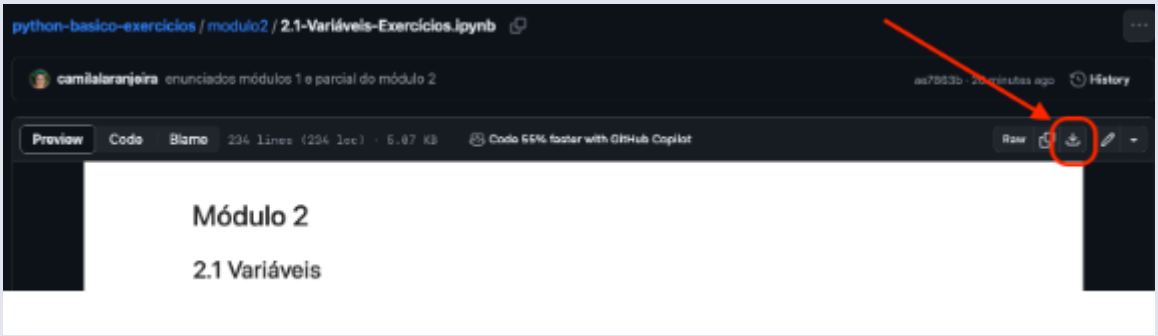
- Palavras reservadas do Python **não** podem nomear variáveis

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

Certo	**spam eggs spam23 _speed max_speed**
Errado	**23spam #sign var.12**
Diferente entre si	**spam Spam SPAM**

Exercício

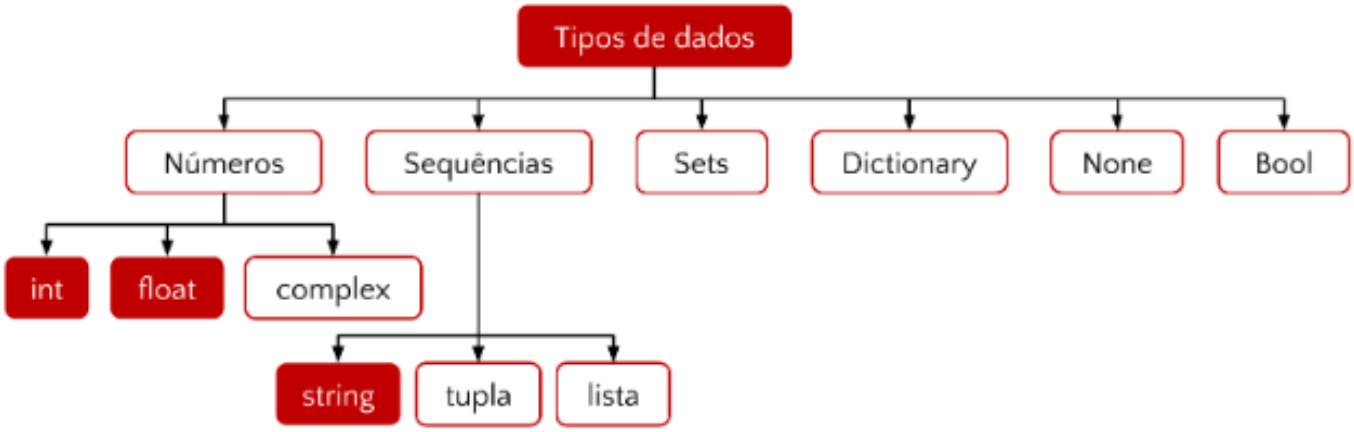
Baixe o arquivo 2.1-Variáveis-Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.



Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo2 do seu repositório.

Tipos de dados e primeiros erros

Já mencionamos que variáveis são compostas por **nome** e **tipo**, sendo os tipos dinamicamente definidos de acordo com o dado armazenado. Na imagem a seguir você encontra os tipos de dados suportados pelo Python, organizados em categorias. Essa variedade de tipos provê a flexibilidade necessária para representar e manipular informações de diferentes naturezas.



Hoje vamos conhecer os tipos essenciais para começar a desenvolver soluções em código:

• Inteiro (int): Representa números inteiros com sinal (negativos, zero, ou positivos), como -1000, -5, 0, 1, 100;
• Ponto Flutuante (float): Representa números decimais com sinal, também chamados de ponto flutuante, como -0.5, 3.14, 2.0;
Pode haver omissão de uma das partes, assume-se zero (ex: -.2, -7.)
• String (str): Representa sequências de caracteres, como "olá!", '123',"oi =)". Sua definição exige que a sequência esteja entre aspas duplas ou simples.

No código a seguir inicializamos três variáveis, atribuindo objetos de diferentes tipos e em seguida imprimimos seus tipos através dos comandos print() e type(). Note que executamos ambos os comandos na mesma linha, sendo o interpretador responsável por executar os comandos mais internos antes dos externos.

tipos.py	terminal
frase = "O nome dela é Jennifer" num = -85 pi = 3.14159265 print(type(frase)) print(type(num)) print(type(pi))	$ python3 tipos.py <class 'str'> <class 'int'> <class 'float'>

Dedicaremos uma aula completa à exploração de strings, o mais robusto dentre os três tipos. No entanto, precisamos conhecê-las para duas operações fundamentais: imprimir mensagens de diversas naturezas na tela, como print('O resultado é: '), além de imprimir mensagens com instruções ao solicitar uma entrada de dados do usuário (veremos em breve).

No Python, até os tipos mais básicos são implementados a partir do paradigma de orientação a objetos, que conheceremos em disciplinas futuras. No entanto, as características básicas desse paradigma nos ajudam a compreender coisas im-

portantes. A palavra "class" que vemos ao consultar o tipo da variável (ex: <class 'int'>) remete às **classes**, nome que se dá à definição dos **dados** e **operações** associados a um determinado tipo de dado.

Usando a classe do tipo float como exemplo, o **dado** a ela associado é o valor numérico (ex.: x = 2.5), e uma de suas **operações** se chama as_integer_ratio (traduzido livremente "como_razão_inteira") que fornece uma representação do par (numerador, denominador) que compõem a fração numérica.

```
>>> x = 2.5
>>> x.as_integer_ratio()
(5, 2)
```

Por isso o conteúdo armazenado nas variáveis (ex: -10, 1.5, "olá") são chamados de **objetos**, já que não são meras representações de dados, mas também dão suporte a uma série de comportamentos aplicáveis ao dado.

Literal / Constante

É importante conhecer a diferença entre literais e variáveis no Python. Nos referimos a literais como valores constantes durante a execução do código. De forma mais simples, é o valor em sua forma original. Na instrução x = 2, temos a variável x associada ao literal 2. Apesar da variável x poder mudar de valor, o literal 2 é sempre constante.

Essa nomenclatura é importante para começarmos o longo processo de entender os erros lançados pelo interpretador. Observe o exemplo a seguir.

```
>>> 2 = x
File "<stdin>", line 1
  2 = x
  ^
SyntaxError: cannot assign to literal
```

O interpretador lança um erro de sintaxe, SyntaxError. Lembrando que a sintaxe se refere à estrutura de escrita das instruções. Ao jogar a frase de erro no tradutor inglês/português, lê-se "não é possível atribuir a literal". Se sabemos o que é uma atribuição e o que é um literal, o erro diz tudo que precisamos saber: que literais não podem receber novos valores, ou seja, não podem estar à esquerda da operação de atribuição.

O que pode dar errado

Ainda sobre erros, vamos falar sobre erros de tipo, ou TypeError. Existem regras para quais operações podem ser aplicadas a cada tipo, e quais tipos podem ser operados juntos. Ao receber do interpretador um erro de tipo, precisamos verificar se a instrução apontada envolve variáveis de tipos suportados pela operação. Um exemplo de erro é apresentado a seguir. É comum para os brasileiros confundir o símbolo que define o float, trocando o ponto pela vírgula.

```
>>> x = 3,7
>>> print(x + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple
```

Em Python x = 3,7 é uma instrução sintaticamente válida, mas não define um float, e sim a tupla (veremos mais à frente na disciplina). Ao tentar fazer uma operação de soma, x+1, o interpretador lança um TypeError. Ainda que você não entenda completamente o erro, por se tratar de um erro de tipo, basta verificar o tipo de todas as variáveis e literais presentes na expressão para identificar nossa confusão ao inicializar incorretamente o x.

```
>>> type(1)
<class 'int'>
>>> type(x)
<class 'tuple'>
```

Já que estamos falando de erros, vou apresentar alguns outros que podemos encontrar no início de nossa jornada. Começando pelo erro de indentação, ou IndentationError.

Indentar se define por adicionar espaços ou tabs no início de uma linha de código para organizar blocos de instruções que integram diferentes estruturas.

Vai fazer mais sentido quando conhecermos algumas estruturas, como laços de repetição ou comandos de decisão, mas no Python a indentação faz parte da sintaxe e alinhamentos inconsistentes lançam erros. Até o momento, conhecemos apenas blocos de instruções de mesmo nível hierárquico, que portanto devem estar com o mesmo alinhamento à esquerda. Observe o seguinte exemplo:

soma.py	terminal

x = 3
print(x)
  x = x + 1
print(x)
	$ python3 soma.py
	File "soma.py", line 3
	  x = x + 1
	IndentationError: unexpected indent

O comando da linha 3 tem uma indentação diferente dos outros, e por isso o interpretador lança o erro que se traduz como: "recuo inesperado". Recuo se refere justamente ao espaçamento no início da linha. Curiosamente os comandos não precisam ter recuo zero, sem nenhum espaço à esquerda, basta que a indentação seja consistente, como no exemplo abaixo onde adicionamos três espaços no início de cada linha.

soma.py	terminal
x = 3 print(x) x = x + 1 print(x)	$ python3 soma.py 3 4

Por fim, vale comentar sobre **erros de lógica**. Nem todo erro vai ser identificado e lançado pelo interpretador. Considere o exemplo a seguir:

media.py	terminal
n1 = 10 n2 = 20 media = n1 + n2 / 2	$ python3 media.py 20

Lendo o código e o nome (bem escolhido 👏) das variáveis, entendemos que o objetivo é calcular a média de dois números. Mas o resultado no terminal está incorreto. Se um programa executa sem lançar erros, mas não exibe o resultado esperado, ele pode conter um erro de lógica, o famoso **bug**. O processo de identificação desses erros se chama **debug**, traduzido como depuração. Aprenderemos a depurar em disciplinas futuras, mas para sanar sua possível curiosidade, o exemplo obedece a regra matemática que divisões têm prioridade sobre somas. A seguir mostramos a execução das instruções escritas incorreta e corretamente.

Execução incorreta:	**media = 10 + 20 / 2 ▶ media = 10+10 ▶ media = 20**
Execução esperada:	**media = (10 + 20)/ 2 ▶ media = 30/2 ▶ media = 15**

Bug: Curiosidade

O termo "bug" vem da época de computadores eletromecânicos. Conta-se que uma mariposa ficou presa em uma das válvulas, causando um erro de execução. A cientista Grace Hopper colou o inseto em seu diário e popularizou o termo na programação.

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula2_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo2 do seu repositório da disciplina no GitHub.

1) Transforme o seguinte algoritmo em código. Lembre-se das funções print e type que já conhecemos de forma geral.

1) Armazene o valor 5 em uma variável e o valor 2 em outra variável
2) Imprima o tipo de dados dessas duas variáveis
3) Divida a primeira variável (com o valor 5) pela segunda variável (com o valor 2) e armazene o resultado em uma terceira variável
4) Imprima a terceira variável e também o seu tipo

2) Transforme o seguinte algoritmo em código. Lembre-se das funções print e type que já conhecemos de forma geral.
1) Armazene o texto "o resultado é" em uma variável, o valor 10 em outra variável, e o valor 3.5 numa terceira variável.
2) Some os valores da segunda e terceira variável e armazene em outra variável.
3) Imprima todas as variáveis na ordem de criação e imprima também seus tipos.

3) Dadas duas variáveis v1 = 10 e v2 = 20, utilize uma terceira variável para trocar os valores entre as duas variáveis. Ou seja, ao final v1 terá o valor de v2, e v2 o valor de v1. Você deve usar uma variável auxiliar de troca, não podendo atribuir os novos valores diretamente.

```
v1 = 10
v2 = 20

Complete o código aqui.

print(v1)
print(v2)
```

4) Uma conta poupança foi aberta com um depósito de R$500,00. Esta conta é remunerada em 1% de juros ao mês. O código a seguir apresenta uma forma de implementação para calcular três meses de acúmulo de juros. Reescreva esse código usando apenas duas variáveis.

Instruções e expressões

Nessa aula vamos distinguir instruções e expressões, para entender o papel de cada uma delas no desenvolvimento. Definimos um programa como um conjunto de instruções que o interpretador executa sequencialmente. Cada **instrução** é uma "frase", ou operação, que a linguagem suporta, como atribuição, impressão, decisão, laço de repetição, entre outras. Cada linha lógica de código representa uma instrução. O termo "linha lógica" é adotado porque podemos dividir a mesma instrução em múltiplas linhas e ela ainda será uma única instrução. Basta para isso adicionar o caractere "\" ao final da linha que se deseja dividir, como apresentado no exemplo a seguir nas linhas 3 e 4.

instrucao.py

```
n1 = 10
n2 = 20
total = n1 * 2 + \
        n2 * 2
print(total)
```

terminal

```
$ python3 instrucao.py
60
```

Instruções são definidas por não terem um retorno significativo, ou seja, realizam ações no programa mas não têm objetivo de gerar um resultado para uso posterior. Em contraste com as **expressões**, que por natureza são executadas para produzir valores como resultado. Instruções podem conter uma ou mais expressões. Por exemplo, na terceira linha lógica do código acima (que ocupa as linhas 3 e 4 do script) temos uma instrução de atribuição contendo uma expressão aritmética composta por somas e multiplicações. Enquanto a expressão produz um resultado, a atribuição executa silenciosamente a associação desse resultado à variável.

Palavra reservada None

Já conhecemos a instrução de impressão na tela, o comando print, apesar de ainda

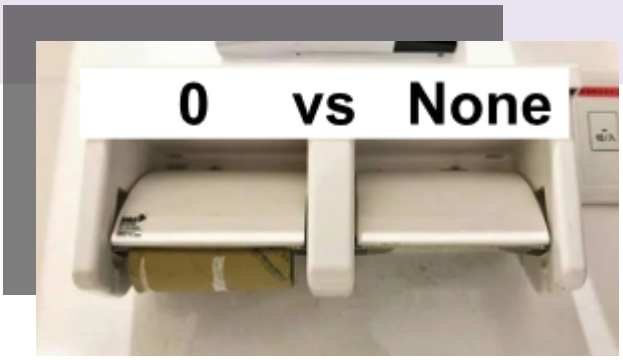
não termos abordado todas as suas possibilidades. Observe o exemplo a seguir, onde tentamos atribuir o resultado do comando print a uma variável e em seguida consultamos esse valor.

```
>>> x = print("Olá")
Olá
>>> x
>>> print(x)
None
```

Ao executar a primeira linha, a ação do comando print é executada, apresentando na linha seguinte a palavra "Olá". Ao consultar o valor de x na segunda linha, variável que recebeu o resultado do print na linha anterior, o python interativo não apresenta nada além dos símbolos ">>>" solicitando um novo comando. Na terceira linha usamos o print para explicitamente visualizar o valor de x, e eis que descobrimos o não-valor, None. Nesse caso significa que nada foi associado à variável x.

valor = None

None é a palavra reservada que representa nulidade, ausência de valor, um conceito comum a diversas linguagens. Pode ser usado para representar explicitamente algo indefinido, por exemplo em uma atribuição:



Instrução de Atribuição

Já conhecemos a instrução de atribuição, a qual definimos como associação de um valor a uma variável, sendo esse valor o resultado de uma expressão. Antes de destrinchar o que define e quais são os tipos de expressão, vale comentar que uma instrução pode possuir múltiplas sintaxes suportadas. No caso da atribuição temos, entre outras:

Simples: atribui um valor a uma variável	>>> a = 10 + 1

Múltipla: atribui o mesmo valor a múltiplas variáveis	>>> a = b = 10 + 1
	>>> print(a)
	11
	>>> print(b)
	11

Sequência: atribui múltiplos valores a múltiplas variáveis. Requer a mesma quantidade de variáveis e expressões de cada lado do operador de atribuição. Note a quantidade de vírgulas.	>>> x, y, z = 10+1, 2, 3
	>>> print(x)
	11
	>>> print(y)
	2
	>>> print(z)
	3

Outras que conheceremos no futuro.	

Expressões

O objetivo de uma expressão é ser avaliada para descobrir o seu retorno/resultado a partir dos operadores e operandos envolvidos. Na expressão de calculadora 10 + 1, por exemplo, temos um operador (+) e dois operandos (os literais 10 e 1).

Nessa disciplina conheceremos principalmente quatro tipos de expressão, sendo esta aula sobre os dois primeiros tipos:

- **Simples**
- **Aritmética**
- Relacional
- Lógica

Usaremos a instrução de atribuição de referência para exemplificar expressões simples e aritméticas, colocando cada expressão exemplificada ao lado direito de uma atribuição.

Expressões simples não possuem operadores. Podem ser:

- Literais/Constantes: seu retorno é o próprio valor. Ex.: a = 10
- Variáveis: seu retorno é o valor do objeto associado à variável. Ex.: b = a
- Chamadas de função: assunto futuro.

Expressões aritméticas possuem operadores aritméticos **binários** e operandos. Ser um operador binário significa que cada operador requer um par de operandos. Seu tipo de **retorno** depende do tipo de dados dos operandos. Usaremos a princípio operandos numéricos (int ou float), que produzem retorno numérico. Esse retorno irá assumir o tipo float se:

- Pelo menos um dos operandos for float
- Em qualquer operação de divisão (/)

Em qualquer outro caso, o retorno será int. Na expressão aritmética, cada operando esperado é uma expressão, sendo a operação na forma:

<expressão 1> **<operador>** <expressão 2>

Na expressão 10 - 5, temos o operador binário de subtração (-) operando duas expressões simples, os literais 10 e 5. Podemos também ter expressões aritméticas compostas, como o exemplo b - 1 + c, envolvendo dois operadores e três operandos. Nesse caso as operações serão executadas **par a par, da esquerda para a direita**. Assumindo b = 5 e c = 2, temos:

b - 1 + c ▶ 5 - 1 + 2 ▶ 4 + 2 ▶ 6

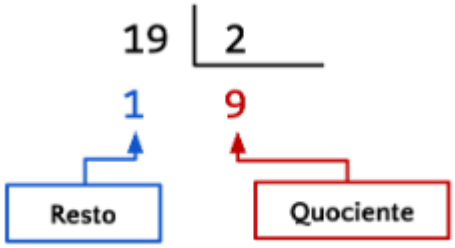
A seguir apresentamos os operadores aritméticos suportados pelo Python, acompanhado de uma instância de cada operador e o resultado da operação.

```
>>> 2 + 2
4
>>> 10 - 4.0
6.0
>>> 2 * 3.5
7.0
>>> 25 / 4
6.25
>>> 25 // 4
6
>>> 25 % 4
1
>>> 2 ** 3
8
```

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
//	Divisão Inteira
%	Módulo (resto)
**	Potência

Enquanto a maioria dos operadores são populares da matemática, vale distinguir operadores de divisão (/), divisão inteira (//) e módulo (%). Considere as seguintes expressões:

```
>>> 19 / 2
9.5
>>> 19 // 2
9
>>> 19 % 2
1
```



Enquanto a divisão (/) fornece o resultado da divisão com parte inteira e parte fracionária, ou seja, como float, a divisão inteira (//) retorna somente a parte inteira da divisão, sendo o resto, também inteiro, retornado pelo módulo (%).

Assim como na matemática, nem toda operação aritmética tem igual prioridade de execução. Na expressão 5 + 2 * 3 por exemplo, a regra dita que a multiplicação deve ser executada antes da soma, ou seja:

5 + 2 * 3 ▶ 5 + 6 ▶ 11

Parênteses podem ser utilizados para alterar a ordem padrão de avaliação. Expressões entre parênteses são executadas primeiro. Por exemplo:

(5 + 2) * 3 ▶ 7 * 3 ▶ 21

O esquema a seguir apresenta a **ordem de precedência de operadores**, organizada de cima para baixo por prioridade. Por exemplo, a potência (**) tem prioridade maior que a soma (+).

```
>>> 1 + 2 ** 3 / 4 * 5
11.0
>>> (1 + 2) ** 3 / 4 * 5
33.75
```

()
**
*, /, %
+, -

Atribuições simplificadas

Em expressões aritméticas da forma

```
a = a + <expressao>
```

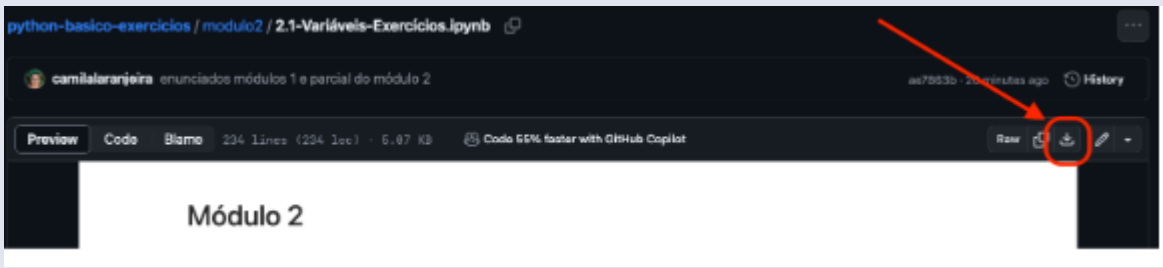
```
a += <expressao>
```

sabendo que a expressão à direita será toda resolvida antes de realizar a última soma e só ao fim será realizada a atribuição. Na tabela abaixo você encontra instâncias de atribuições simplificadas com cada um dos operadores que acabamos de conhecer.

Operador	Exemplo	Corresponde a
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
//=	a //= b	a = a // b
%=	a %= b	a = a % b
**=	a **= b	a = a ** b

Exercício

Baixe o arquivo 2.3-Instruções-Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.



Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo2 do seu repositório.

Comentários & funções básicas: input e print

Para implementar soluções mais completas, precisamos receber dados de entrada do usuário, e apresentar na tela a saída do processamento. Para isso temos as funções input e print.

Input()

Podemos instruir o Python a pausar a execução e aguardar uma entrada de dados do usuário. Se você digitar no python interativo o comando input(), na linha seguinte onde seria apresentado o resultado da execução, o interpretador irá aguardar uma entrada de dados

```
>>> x = input()
```

Só após digitar um valor qualquer e pressionar a tecla enter, o terminal apresenta o símbolo ">>>" aguardando novo comando. Observe a sequência de comandos a seguir.

```
>>> x = input()
3
>>> print(x)
3
>>> type(x)
<class 'str'>
>>>
```

Na primeira linha solicitamos uma entrada de dados e atribuímos à variável x. Na segunda, o valor está colorido de laranja para indicar que se trata de uma entrada de dados fornecida para o programa. Em seguida consultamos o valor de x, confirmando nossa entrada. Por fim, mas não menos importante, consultamos o tipo da variável, descobrindo que se trata de uma string.

Os dados recebidos através do comando input() **são sempre do tipo string**. O código a seguir apresenta um script mais completo de entrada de dados. Nele vemos também que dentro dos parênteses do input() podemos adicionar uma mensagem que será apresentada na tela para o usuário antes de solicitar a entrada. Lembrando que as entradas de dados estão coloridas em laranja, enquanto as impressões do código permanecem em branco.

dados.py	terminal
nome = input('Qual o seu nome?') print(type(nome)) idade = input('Qual a sua idade?') print(type(idade))	$ python3 dados.py Qual o seu nome? Jose <class 'str'> Qual a sua idade? 17 <class 'str'>

Já falamos que cada tipo de dado é tratado de maneira distinta pelo interpretador. Portanto, dados recebidos pelo input() **precisam ter seu tipo convertido**. As palavras reservadas que nomeiam tipos são também comandos de conversão. No código acima, para usar a idade como um valor inteiro (por exemplo em uma expressão aritmética), precisamos convertê-la usando o comando int() e alimentando dentro dos parênteses a variável a ser convertida.

dados.py	terminal
idade = input('Qual a sua idade?') print(type(idade)) idade = int(idade) print(type(idade)) print("Você nasceu em") print(2024-idade)	$ python3 dados.py Qual a sua idade? 17 <class 'str'> <class 'int'> Você nasceu em 2007

Caso a conversão não seja realizada, receberemos um erro de tipo, TypeError, já que a operação aritmética não opera strings com inteiros. Entenderemos melhor na aula de strings.

dados.py	terminal
idade = input('Qual a sua idade?') print("Você nasceu em") print(2024-idade)	$ python3 dados.py Qual a sua idade? 17 TypeError: unsupported operand type(s) for -: 'int' and 'str'

O tipo de conversão depende da necessidade imposta pelo dado. Note, no exemplo a seguir, que executamos ambas a entrada de dados e a conversão de tipo numa mesma linha, sendo os comandos dos parênteses mais internos executados antes dos externos.

dados.py	terminal
idade = int(input('Qual a sua idade?')) peso = float(input('Qual o seu peso?'))	$ python3 dados.py Qual a sua idade? 17 Qual o seu peso? 63.5

Print()

O print é o primeiro comando aprendido por qualquer pessoa aprendendo a programar!

ola.py	terminal
print("Olá mundo!")	Olá mundo!

Note que o print resolve a expressão entre parênteses, removendo as aspas ao apresentar a saída. Se usamos o Python interativo para consultar o valor de uma string sem usar o print, o valor permanece entre aspas, já que elas fazem parte do dado.

```
>>> frase = "Olá mundo!"
>>> frase
"Olá mundo!"
>>> print(frase)
Olá mundo!
```

A impressão em tela também permite concatenar múltiplas informações separadas por vírgula, não limitadas a literais do tipo string. Podemos alimentar variáveis e literais de quaisquer tipos, além de outras expressões.

```
>>> x = 10
>>> print(x, x*2)
10 20
>>> print("O dobro de", x, "é", x * 2)
O dobro de 10 é 20
```

Note que o resultado do comando print(x, x*2) apresenta os valores 10 e 20 separados por um espaço em branco, além de adicionar uma quebra de linha ao final da expressão. Temos controle sobre esses elementos, podendo alterá-los dentro do print atribuindo novos valores às variáveis de caractere separador (sep) e caractere de finalização de linha (end).

```
>>> x = 10
>>> print(x, x*2, sep=", ", end=" ")
10, 20 >>>
```

No exemplo acima definimos, dentro do próprio print, a sequência de caracteres ", " (vírgula e espaço em branco) como separador de valores, e o caractere " " (espaço em branco) para finalizar a linha. Esse último faz com que os símbolos ">>>"

aguardando um novo comando apareçam na mesma linha da impressão.

Também podemos realizar expressões de **formatação de strings**, permitindo a inclusão de expressões. O código a seguir apresenta algumas das possíveis sintaxes

```
>>> nome = input()
Camila
>>> idade = int(input())
30
>>> print(f"Meu nome é {nome}, tenho {idade} anos.")
Meu nome é Camila, tenho 30 anos.
>>> print('Meu nome é %s, tenho %d anos.' % (nome, idade))
Meu nome é Camila, tenho 30 anos.
>>> print('Meu nome é {}, tenho {} anos.'.format(nome, idade))
Meu nome é Camila, tenho 30 anos.
```

Em ambientes de desenvolvimento como VSCode ou Jupyter, caracteres de formatação de strings são coloridos de maneira distinta do restante da string, diferenciando assim caracteres impressos literalmente como aparecem no código das expressões que ainda serão resolvidas.

A primeira das sintaxes apresentada acima, a f-string ("f" de formatada), é a forma mais legível e concisa de formatação. Consiste em adicionar a letra "f" antes do primeiro sinal de aspas, e usa a sintaxe de chaves ({}) para inserir expressões no meio da string.

Esse post do Real Python traz um detalhamento das 3 sintaxes. A partir daqui falaremos somente das f-strings. Lembre-se que você pode traduzir a página no navegador.

```
>>> kmh = int(input("Digite uma velocidade em km/h: "))
Digite uma velocidade em km/h: 80
>>> print(f"{kmh}km/h é {kmh/3.6}m/s")
80km/h é 22.22222222222222m/s
```

Note no código acima que a conversão de 80 quilômetros por hora em metros por segundo produz uma dízima periódica e é apresentada na tela com 14 caracteres na parte decimal. As f-strings também permitem controlar o resultado de expressões através de seu protocolo de formatação. Referido na própria documentação como "mini-linguagem de especificação de formato", esse protocolo é complexo e robusto o suficiente para receber o status de "mini-linguagem". Veremos aqui apenas uma demonstração de suas capacidades.

A sintaxe completa da f-string utiliza {} (chaves) e um separador : (dois pontos) para representar tanto a expressão quanto a formatação da saída: "{expressão:formatação}"

Aplicando o protocolo de formatação em inteiros:

```
>>> num = 123456
>>> print(f"Número com 10 caracteres: {num:10}")
Número com 10 caracteres:     123456
>>> print(f"Número com 10 caracteres precedido por 0: {num:010}")
Número com 10 caracteres precedido por 0: 0000123456
>>> print(f"Número com vírgula separando o milhar: {num:,}")
Número com vírgula separando o milhar: 123,456
>>> print(f"Número com 10 caracteres e vírgula separando o milhar: {num:10,}")
Número com 10 caracteres:     123,456
```

Aplicando o protocolo de formatação em floats, vale explicar. A formatação completa tem a sintaxe "x.yf", sendo x o número de caracteres da parte inteira e y o número de caracteres parte decimal de um número, seguidos pelo caractere "f" de float. No exemplo a seguir definimos que a conversão em m/s terá 2 caracteres na parte inteiro e 0 na parte decimal:

```
>>> print(f"{80}km/h é {80/3.6:2.0f}m/s")
80km/h é 22m/s
```

Um uso comum de formatação é imprimir valores monetários, adotando por exemplo a vírgula como separador de milhar, e limitando a parte decimal a 2 caracteres. Dessa maneira:

```
>>> print(f"Seu saldo é R${123456:,.2f}")
Seu saldo é R$1,234.56
```

Note que omitimos o número de caracteres da parte inteira, que será definido dinamicamente de acordo com o número sendo impresso.

Comentários

Em Python, comentários são utilizados para incluir anotações ou explicações no código fonte. Comentários não são executados durante a execução do programa e são completamente ignorados pelo interpretador. Eles são úteis para documentar o código, explicar a lógica por trás de determinadas decisões, ou para fornecer informações adicionais para quem lê o código. Existem dois tipos principais de comentários em Python:

Comentários de linha única: são iniciados pelo caractere # e continuam até o final da linha.

```
# Este é um comentário de linha única
# variavel = int(input())
# o interpretador não vai executar a linha acima
variavel = 10  # Posso comentar no final de uma linha de código
```

Comentários de múltiplas linhas: São delimitados por três aspas simples (''') ou três aspas duplas ("""). Podem ser usados para comentários extensos.

```
'''
Este é um comentário de múltiplas linhas.
Ele abrange várias linhas e é delimitado por três aspas simples.
'''
print('Olá')
```

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula4_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo2 do seu repositório da disciplina no GitHub.

1) Faça um programa para ler as dimensões de um terreno em inteiros (comprimento e largura), bem como o preço do metro quadrado da região em ponto flutuante. Calcule o valor do terreno e imprima o resultado como mostra o exemplo a seguir:

O terreno possui 250m2 e custa R$512,490.50

Comente na linha acima de cada instrução uma breve descrição da instrução.
Fórmulas:
- area_m2 = comprimento * largura
- preco_total = preco_m2 * area_m2

2) Leia um valor inteiro correspondente a uma temperatura em graus Fahrenheit e apresente-a convertida em graus Celsius.

Fórmula de conversão: C = (F - 32) * (5/9), sendo C o valor em graus Celsius e F o valor em Fahrenheit. Antes de imprimir, **converta o valor em Celsius** para inteiro. A mensagem deve estar formatada da seguinte maneira:
86 graus Fahrenheit são 30 graus Celsius.

3) Você está desenvolvendo um programa para calcular o preço total de uma compra em uma loja online. O preço dos produtos é calculado multiplicando a quantidade pelo preço unitário. Escreva um programa em Python que solicite do usuário o nome, o preço unitário e a quantidade de 3 produtos comprados e imprima ao final o preço total da compra. Note no exemplo a seguir que:

-Cada entrada de dados tem uma mensagem indicando o dado solicitado (mensagens em itálico, dado de entrada em negrito)
-A saída deve estar formatada exatamente como indicado (a string "Total: R$" e o preço com um separador de milhar e duas casas decimais).

Entrada	Saída
Digite o nome do produto 1: **calça** Digite o preço unitário do produto 1: **199,90** Digite a quantidade do produto 1: **3** Digite o nome do produto 2: **camisa** Digite o preço unitário do produto 2: **49,95** Digite a quantidade do produto 2: **10** Digite o nome do produto 3: **cinto** Digite o preço unitário do produto 3: **25** Digite a quantidade do produto 3: **3**	Total: R$1,174.20

4) Escreva um programa que leia um valor inteiro referente a uma quantia em reais e calcule a menor quantidade possível de notas necessárias para pagar aquele valor. As notas possíveis são: 100, 50, 20, 10, 5, 2, 1. **A saída deve estar formatada exatamente como indicado.**

Entrada	Saída
576	5 nota(s) de R$100,00 1 nota(s) de R$50,00 1 nota(s) de R$20,00 0 nota(s) de R$10,00 1 nota(s) de R$5,00 0 nota(s) de R$2,00 1 nota(s) de R$1,00

Condicionais

Condicionais

Aprendemos no módulo anterior que o papel das expressões é ser avaliada para descobrir o seu retorno/resultado. Nesse módulo vamos conhecer as duas expressões remanescentes

·Simples

·Aritmética

·**Relacional**

·**Lógica**

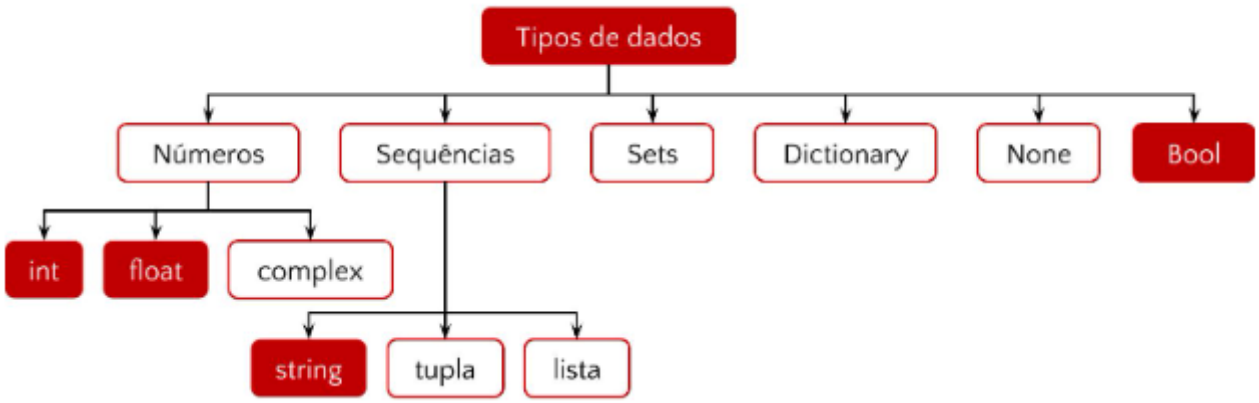
Álgebra Booleana

Definida na obra "Uma investigação das leis do pensamento" pelo matemático e lógico George Boole, a álgebra booleana busca representar uma versão simplificada do pensamento humano através de expressões matemáticas. Nessa teoria, expressões possuem um **"valor verdade"** binário, ou seja, tem dois valores possíveis que representam a validade de uma expressão:

True: verdadeiro

False: falso

Apelidada de lógica "booleana", em referência ao seu propositor George **Boole**, se tornou fundamental na computação, incorporando estruturas de controle de fluxo e tomadas de decisão em linguagens de programação. Introduz um novo tipo de dado, chamado bool:



True e False são palavras reservadas do Python representando os possíveis estados de uma dado booleano, seja ele um literal, uma variável ou retorno de expressão.

```
>>> print(type(True))
<class 'bool'>
>>> x = False
>>> print(type(x))
<class 'bool'>
```

Lembrando que palavras reservadas são interpretadas pelo interpretador (desculpe a redundância) e assumem comportamentos específicos em expressões. Valores booleanos possuem equivalentes numéricos, com False equivalente a 0 (zero) e True valendo 1 (um).

S1M NÃ0

```
>>> print(1 + True)
2
>>> print(10/False)
ZeroDivisionError: division by zero
```

No entanto, ao converter outros tipos de dados para o tipo bool, a conversão retornará False apenas para o valor numérico zero (int ou float). Qualquer outro valor será convertido para True.

```
>>> bool(9)
True
>>> bool(-1)
True
>>> bool("ola")
True
>>> bool("0")
True
>>> bool(0)
False
>>> bool(0.0)
False
```

Expressões Relacionais

A expressão relacional é uma comparação entre duas expressões, que retorna um valor booleano, sendo:

•Zero (0) ou False se a relação é falsa

•Um (1) ou True se a relação é verdadeira

Por envolver duas expressões, seus operadores são binários.

<expressão 1> <operador> <expressão 2>

As expressões relacionais em programação são similares às comparações matemáticas e são usadas para avaliar relações entre valores. A analogia com a matemática é bastante direta:

Símbolo	Significado	<exp1><símbolo><exp2>
<	menor que	a<b
<=	menor ou igual que	a + 1 <= 10
==	igual	a=="Setembro"
>=	maior ou igual que	a>=b*2
>	maior que	a>27.5
!=	diferente	a+b!=c-d

Mas **cuidado!** Não confunda o operador de atribuição (=) Com o operador relacional de igualdade (==)

```
>>> a, b = 10, 5 * 2
>>> print(a == b) # RELAÇÃO
True
>>> print(a = b)  # ATRIBUIÇÃO
TypeError: 'a' is an invalid keyword
argument for print()
```

Sobre precedência de operadores, expressões aritméticas têm prioridade sobre as relacionais. Mas entre si, operadores relacionais têm igual precedência, ou seja, a ordem de execução segue o padrão da esquerda para a direita. Por exemplo:

( )
**
*, /, %
+ , -
>, <, >=, <=, ==, !=

2 + 5 > 10 * 2 == True
7 > 20 == True
False == True
False

Atenção aos tipos! Algumas relações não são suportadas pela linguagem. No exemplo a seguir, recebemos uma entrada com o comando input, que já sabemos sempre retornar string. Ao avaliar a expressão relacional menor que (<) entre uma string e um inteiro, o interpretador lança um TypeError. Já a relação de igualdade (==) pode ser avaliada entre tipos distintos, no entanto, o retorno da igualdade é False, e esse pode não ser o comportamento desejado.

```
>>> idade = input()
18
>>> idade
"18"
>>> print(idade < 18)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
>>> print(idade == 18)
False
```

Expressões Lógicas

Expressões lógicas estão inseridas no mesmo contexto da álgebra de boole, que introduz os operadores lógicos and, or e not. **Os operandos envolvidos em expressões lógicas devem necessariamente ser expressões com valor booleano**, sejam elas literais, variáveis ou retornos de expressão. Conheceremos a seguir cada um dos operadores.

·**and** (**E** lógico): Operador binário que retorna True somente se ambas as condições associadas forem verdadeiras; em qualquer outro caso, retorna False.

```
>>> a = 5
>>> b = 10
>>> (a > 0)  and (b < 20)  # True and True
True
>>> (a == 0) and (b < 20)  # False and True
False
>>> (a <= 5) and (b != 10) # True and False
False
>>> (a < 2)  and (b > 10)  # False and False
False
```

·**or** (**OU** lógico): Operador binário que retorna True se pelo menos uma das condições associadas for verdadeira; retorna False somente se ambas forem falsas.

```
>>> a = 5
>>> b = 10
>>> (a > 0)  and (b < 20)  # True and True
True
>>> (a == 0) and (b < 20)  # False and True
False
>>> (a <= 5) and (b != 10) # True and False
False
>>> (a < 2)  and (b > 10)  # False and False
False
```

- **not** (**NÃO** lógico): Operador **unário**, ou seja, que atua sobre um único operando. Inverte o valor de uma expressão, ou seja, se a expressão original tem valor True, not a tornará False, e vice-versa.

```
>>> a = 5
>>> b = 10
>>> not (a == 5) # not True
False
>>> not (b > 20) # not False
True
```

Sobre precedência de operadores, expressões lógicas são as de menor prioridade, mas **note que não estão no mesmo nível de prioridade entre si.**

```
>>> a, b, c = -5, 10, 0
>>> print(b > 0 or not a > 0 and c > 0)
True
# 10 > 0 or not -5 > 0 and 0 > 0
# True or not False and False
# True or True and False
# True or False
# True
```

( )
**
*, /, %
+ , -
>, <, >=, <=, ==, !=
not
and
or

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula2_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo3 do seu repositório da disciplina no GitHub

1) Juliana e Cris querem entrar em um bar, mas só podem se ambos forem maiores de idade (>17). Escreva um programa que solicite as idades de Juliana e Cris e imprima True se ambas forem maiores de 17 anos, indicando que podem entrar no bar, e False caso contrário.

2) Dando continuidade à questão anterior, um outro bar permite a entrada de grupos onde pelo menos uma pessoa é maior de idade (ficando responsável pelas outras). Ajuste sua resposta da questão anterior, ainda solicitando as idades de Juliana e Cris, mas ajustando a expressão para esse novo cenário, imprimindo True se puderem entrar no bar, e False caso contrário.

3) Você está desenvolvendo um sistema de admissão para um clube juvenil de jogos de tabuleiro. Escreva um programa em Python que pergunte ao usuário sua idade, se já jogou pelo menos 3 jogos de tabuleiro (resposta deve ser True ou False) e quantas vezes venceu um jogo. O programa deve imprimir True se o participante tiver entre 16 e 18 anos, já tiver jogado pelo menos 3 jogos e já ter vencido pelo menos 1 jogo, permitindo seu ingresso no clube. Sua expressão deve imprimir False caso contrário. Aqui está um exemplo de interação com seu código no terminal, com entradas de dados destacadas em laranja e as impressões de seu código em branco.

```
Terminal
Digite sua idade: 17
Já jogou pelo menos 3 jogos de tabuleiro? True
Quantos jogos já venceu? 2
Apto para ingressar no clube de jogos de tabuleiro: True
```

4) Você é mestre de uma mesa de RPG e vai criar um sistema para validar uma ficha de personagem. Cada personagem tem uma classe específica com requisitos de atributos. Escreva um script que solicita a classe de personagem escolhida (guerreiro, mago ou arqueiro), os pontos de força e os pontos de magia atribuídos ao personagem. O programa deve imprimir True se os pontos de atributo são consistentes com a classe escolhida, seguindo as seguintes regras:

Guerreiro: Força deve ser igual ou superior a 15, Magia deve ser 10 ou menos.
Mago: Força deve ser 10 ou menos, Magia deve ser igual ou superior a 15.
Arqueiro: Força e Magia devem ser ambos superiores a 5, mas nenhum deles pode ser superior a 15.

O programa deve imprimir False se os pontos de atributo não são consistentes com a classe escolhida. Segue um exemplo de interação com seu código no terminal, com entradas de dados destacadas em laranja e as impressões de seu código em branco.

```
Escolha a classe (guerreiro, mago ou arqueiro): mago
Digite os pontos de força (de 1 a 20): 8
Digite os pontos de magia (de 1 a 20): 17
Pontos de atributo consistentes com a classe escolhida: True
```

5)Solicite de um usuário seu gênero ("M" ou "F"), sua idade e seu tempo de serviço (em anos) e **escreva uma expressão** que imprima True se a pessoa já pode se aposentar, ou False caso contrário, de acordo com as seguintes regras:

A: Para mulheres, ter mais de 60 anos. Para homens, 65.
B: Ou ter trabalhado pelo menos 30 anos
C: Ou ter 60 anos e trabalhado pelo menos 25.

Tabela Verdade

Expressões lógicas são usadas para validar condições necessárias para determinados eventos. Um dos principais desafios de propor expressões lógicas para um dado problema é cobrir todas as suas possibilidades.

A tabela verdade é uma representação tabular de todas as combinações possíveis de valores-verdade para uma expressão lógica. Ela ajuda a visualizar como a expressão se comporta para diferentes valores de entrada booleana.

Cada expressão (lógica ou relacional) só pode assumir dois valores, True ou False. A título de exemplo, considere a tabela verdade que mapeia todas as possibilidades da expressão <exp1>: ano < 2000. Ela terá duas linhas, uma para o caso True e outra para o False.

Ex:
<exp1>: ano < 2000

<exp1>
True
False

Duas expressões criam 4 possibilidades de combinações.

Ex:
<exp1>: ano < 2000
<exp2>: mes == "maio"

<exp1>	<exp2>
True	True
False	True
True	False
False	False

O número de possibilidades vai dobrar a cada nova expressão, é uma relação de 2n possibilidades, sendo n o número de expressões. Mas a gente não precisa se preocupar com isso agora :)

Ao combinar expressões, a tabela verdade nos ajuda a visualizar todos os possíveis valores do nosso problema e as possíveis saídas de uma dada expressão composta.

·Operação and

<exp1>	<exp2>	<exp1> and <exp2>	Ex: de código de saída
True	True	True	ano = 1993 mes = "abril" print(ano < 2000 and mes == "maio")
False	True	False	
True	False	False	
False	False	False	False

·Operação or

<exp1>	<exp2>	<exp1> and <exp2>	Ex: de código de saída
True	True	True	ano = 1993 mes = "abril" print(ano < 2000 and mes == "maio")
False	True	True	
True	False	True	
False	False	False	True

·Operação not

<exp1>	not <exp1>	Ex: de código de saída
True	False	ano = 1993 mes = "abril" print(not ano < 2000)
False	True	True

Tabelas-verdade são especialmente úteis em expressões mais longas, permitindo quebrá-las em expressões menores. Em geral, é um recurso visual útil para operações da álgebra booleana, sendo cada linha é uma combinação possível de valores-verdade para o conjunto de expressões e cada colunas uma expressão ou operação com expressões.

a	b	not a	not b	not a or not b
True	True	False	False	False
False	True	True	False	True
True	False	False	True	True
False	False	True	True	True

Condicionais

Expressões lógicas são usadas para validar condições necessárias para determinados eventos. Em termos práticos, alguns trechos de código só devem ser executados quando determinadas condições são atendidas. Essa é a primeira estrutura que conheceremos capaz de alterar o fluxo linear de execução do código, produzindo desvios de fluxo,

A estrutura condicional começa com a palavra-chave if, seguida pela condição que será avaliada, uma expressão com resultado booleano. O símbolo ":" (dois pontos) que finaliza a linha é obrigatório de acordo com a sintaxe do comando. Quaisquer comandos internos ao if serão executados somente se a expressão retornar True. A **indentação** é crucial na sintaxe das estruturas condicionais, já que o **bloco de comandos** internos a um escopo é determinado pelo seu recuo à esquerda. A tecla Tab é comumente utilizada no início para adicionar o **recuo**. Observe o seguinte código, onde setas representam o uso da tecla Tab:

```
if <expressão>:
→ #comando interno 1
→ #...
→ #comando interno n
# comando externo
```

No exemplo, os comandos internos 1 a n estão condicionados ao resultado da <expressão>. Caso ela seja falsa, o fluxo de execução ignora todos os comandos internos e parte para o próximo comando externo. Note que reduzimos o recuo para finalizar o bloco condicional e escrever o próximo comando externo. É importante destacar que o uso de tabs ou espaços em branco para adicionar um recuo é muito particular de cada editor de texto e até de cada pessoa programando, o único requisito essencial é **que o espaçamento seja consistente!** Ou seja, se o início de um bloco utiliza o tab como recuo, todo o bloco deve utilizar o mesmo recuo.

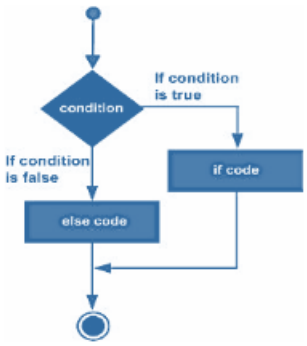
Seguem alguns exemplos práticos de estruturas condicionais com blocos internos de uma ou mais instruções, bem como o resultado da sua execução. Caso uma condição seja falsa, seu bloco interno de comandos não é executado. Já os comandos intermediários (print("Fim")) executam independente do resultado da expressão anterior.

script.py	terminal

x, y = -5, 10
if y > x: # Verdadeiro
  print("y é maior que x")
print("Fim 1")
if x > 0: # Falso
  print("x é positivo")
print("Fim 2")
if x < 0 and y > 0: # Verdadeiro
  print("x é negativo")
  print("y é positivo")
print("Fim 3")
	y é maior que x
	Fim 1
	Fim 2
	x é negativo
	y é positivo
	Fim 3

Há casos onde precisamos executar uma ação caso a condição seja verdadeira, ou uma ação específica caso seja falsa. Para tal, criamos um ramificação com a palavra reservada else.

```
if <expressao>:
        # comandos se a
        # expressão for
        # verdadeira
else:
        # comandos se
        # for falsa
# comandos externos que
# não dependem da expressão
```



Trabalhando com exemplos práticos, considere o seguinte código:

script.py	terminal

x = int(input())
if x > 0:
  print("condição é verdadeira")
  print("x é positivo")
else: *# not x > 0*
  print("condição é falsa")
  print("x é zero ou negativo")
print("Fim")
	-5
	condição é falsa
	x é zero ou negativo
	Fim

O else necessariamente estará associado a um if que o precede. Implicitamente, ele é a negação da condição testada no if.

Operador Ternário

O operador ternário em Python é uma forma condensada de expressar uma estrutura condicional em uma única linha. É uma maneira concisa e elegante de escrever expressões condicionais, proporcionando uma sintaxe mais enxuta para situações simples de decisão. A estrutura básica do operador ternário é a seguinte:

<valor_se_verdadeiro> if <condicao> else <valor_se_falso>

Nesta expressão, condicao é a condição que será avaliada. Se a condição for verdadeira, a expressão retorna valor_se_verdadeiro; caso contrário, retorna valor_se_falso. Assim como outras expressões, podemos utilizar dentro de prints, em atribuições, ou qualquer instrução que capture seu retorno. Aqui estão alguns exemplos práticos:

script.py	terminal
a, b = int(input()), int(input()) print("a é maior" if a>b else "b é maior")	5 7 b é maior
a, b = int(input()), int(input()) maior = a if a > b else b print(maior, "é maior")	5 7 7 é maior

script.py	terminal
a, b = int(input()), int(input())	5
print(a) if a>b else print(b)	7
	7

Em resumo, condicionais produzem desvios de fluxo validando condições que retornam um valor booleano (ex: expressões relacionais ou lógicas). A indentação/recuo/tabulação define o "escopo" de comandos dentro de cada condicional. Além disso, é possível realizar expressões condicionais que em uma linha testam uma condição e produzem um retorno.

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula4_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo3 do seu repositório da disciplina no GitHub.

1)Escreva um programa que lê dois números e informa se a sua soma é par ou ímpar. Critério: se o resto da divisão do número por 2 for 0, o número é par, caso contrário é ímpar. Lembre-se do operador do python % que retorna o resto de uma divisão.

2)Você está criando um sistema de classificação de filmes com base nas avaliações dos usuários. Escreva um programa em Python que solicita ao usuário para inserir a avaliação de um filme em uma escala de 1 a 5. O programa deve imprimir uma mensagem correspondente à classificação do filme:
Se a avaliação for 5, imprima "Excelente!"
Se a avaliação for 4, imprima "Muito Bom!"
Se a avaliação for 3, imprima "Bom!"
Se a avaliação for 2, imprima "Regular."
Se a avaliação for 1, imprima "Ruim."

3)Você está desenvolvendo um programa para verificar se um ano é bissexto. Escreva um código em Python que solicita ao usuário para inserir um ano e imprime "Bissexto" se o ano for (1) divisível por 4 e não for divisível por 100, ou (2) se for divisível por 400. Caso contrário, imprima "Não Bissexto".

Teste seu código com os valores: 1900 (não bissexto), 2000 (bissexto), 2016 (bissexto) e 2017 (não bissexto).

4)Você está implementando um sistema de entrega expressa e precisa calcular o valor do frete com base na distância e no peso do pacote. Escreva um código que solicita a distância da entrega em quilômetros e o peso do pacote em quilogramas. O programa deve **calcular e imprimir o valor do frete** de acordo com as seguintes regras:

Distância até 100 km: R$1 por kg.
Distância entre 101 e 300 km: R$1.50 por kg.
Distância acima de 300 km: R$2 por kg.
Acrescente uma taxa de R$10 para pacotes com peso superior a 10 kg

Decisões Múltiplas

Nem sempre as estruturas de decisão terão somente duas possibilidades. Considere um caso simples, um programa que recebe um número de matrícula e imprime o nome do aluno correspondente. Com o que sabemos até agora teríamos que construir uma sequência de testes de condição usando somente o if.

```
matricula = int(input())
if(matricula == 10129):
   print("Alberto Roberto\n")
if(matricula == 12308):
   print("Bruna Cândida\n")
...
if(matricula == 11908):
   print("Zidane Silva\n")
```

O principal problema disso é que mesmo quando a condição correta for encontrada (vamos supor que no exemplo acima recebemos como entrada a matrícula do Alberto Roberto), todas as condições seguintes serão testadas mesmo assim. Existe no Python a ramificação elif (junção das palavras else e if) que permite criar uma cadeia de decisões múltiplas, interrompendo os testes condicionais **assim que encontra a primeira condição verdadeira.**

```
matricula = int(input())
if(matricula == 10129):
   print("Alberto Roberto\n")
elif(matricula == 12308):
   print("Bruna Cândida\n")
...
elif(matricula == 11908):
   print("Zidane Silva\n")
else:
   print("Erro: matrícula não registrada")
```

Uma cadeia de decisão múltipla pode **opcionalmente** ser finalizada com uma cláusula else. Isso indica que nenhuma das condições daquela cadeia foi verdadeira.

Formalizando a estrutura de decisões múltiplas, temos:

```
if <condicao 1>:
        # comandos executados se a condição 1 é verdadeira
elif <condicao 2>:
        # comandos executados se a condição 1 é falsa e a
        #      condição 2 é verdadeira
else:
        # comandos se as condições anteriores são falsas
```

Um comportamento interessante demonstrado [nesse tutorial de condicionais do Real Python](#) é apresentado a seguir. Na cadeia de decisões a seguir, a segunda expressão contém uma divisão por zero e a terceira faz referência a uma variável indefinida var. Qualquer um dos dois geraria um erro, mas nenhum deles será avaliado porque a primeira condição especificada é verdadeira.

```
>>> var  # Não definida
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    var
NameError: name 'var' is not defined
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
>>> if 'y' in 'Python':
...     print('Primeiro')
... elif 1/0:
...     print("Segundo")
... elif var > 0:
...     print("Terceiro")
...
Primeiro
```

Condições Aninhadas

Condicionais múltiplas podem ser **aninhadas** em qualquer profundidade. Em termos práticos, significa que podemos criar blocos condicionais dentro de blocos condicionais quantas vezes for necessário. Idealmente não é interessante criar tantos blocos internos por reduzir a legibilidade, dificultar a manutenção do código, entre outros problemas. Mas é perfeitamente comum criar dois ou três níveis de blocos internos, como formalizado a seguir.

```
if <condicao 1>:
        # comandos executados se a condição 1 é verdadeira
        if <condicao 2>:
                # comandos executados se as condições 1 e 2
                # são verdadeiras
        else:
                # comandos executados se a condição 1 é
                # verdadeira e a condição 2 é falsa
else:
      # comandos executados se a condição 1 é falsa
```

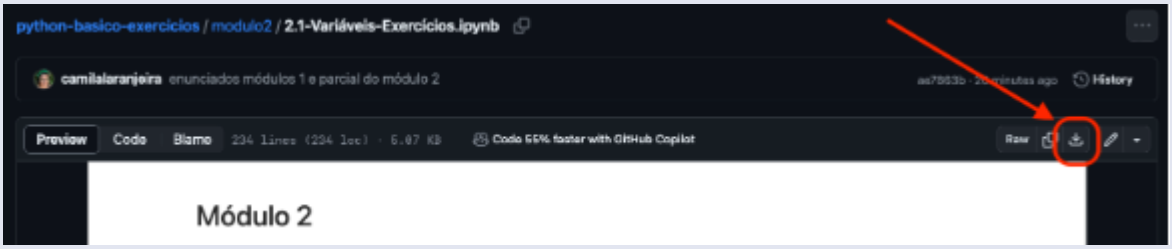
Considere o exemplo prático a seguir:

```
num = input("Digite um numero: ")
if num == 10:
        print("Voce acertou!")
else:
        print("Você errou!")
        if num > 10:
                print("O numero é maior que 10.")
        else:
                print("O numero é menor que 10.");
```

Quando criamos decisões múltiplas, seja pelo elif ou por condicionais aninhadas, precisamos rastrear o resultado das condições anteriores ao longo da execução. O aninhamento só passa para o próximo nível se a condição anterior for verdadeira.

Exercício

Baixe o arquivo 3.5-Decisões_Múltiplas-Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.



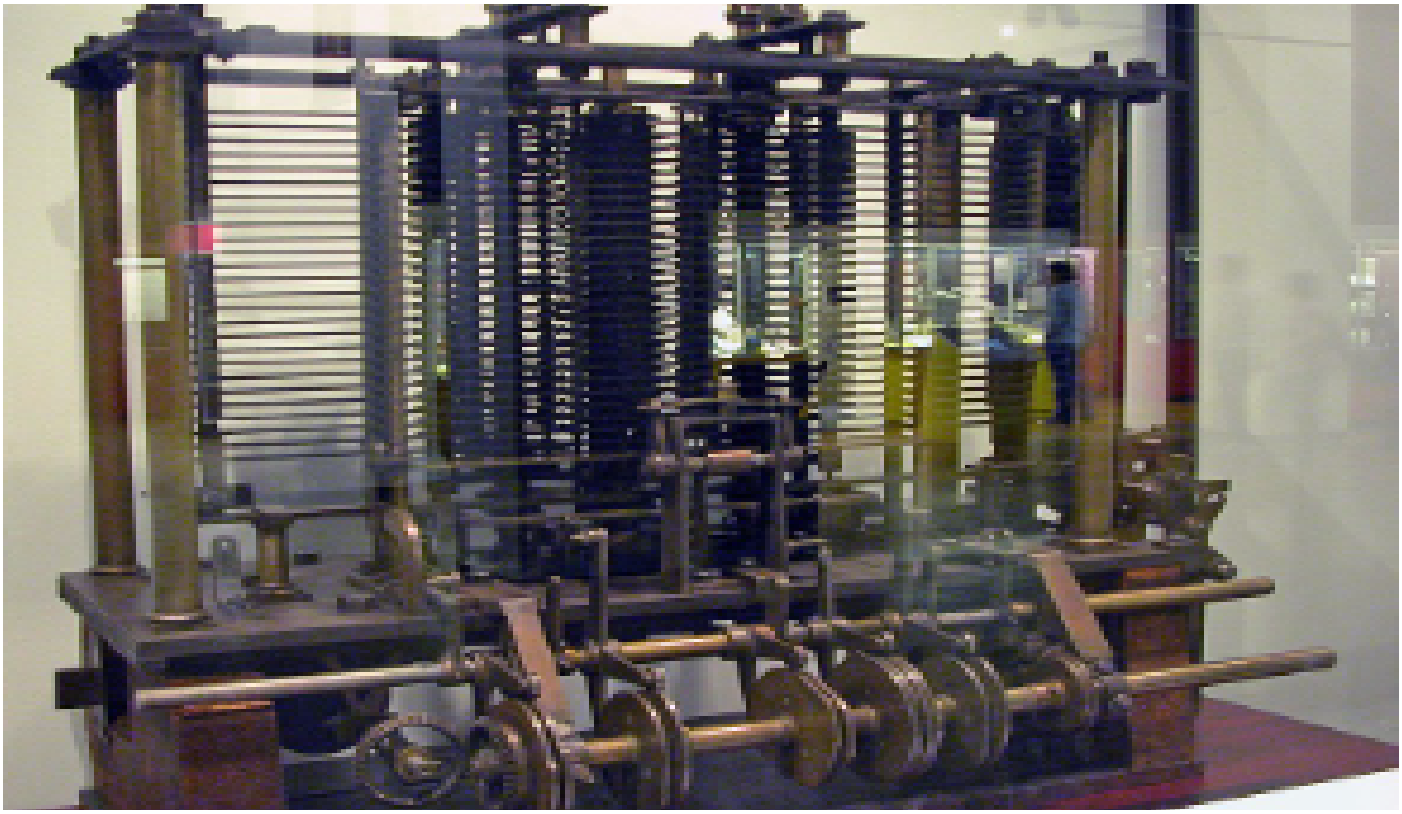
Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo3 do seu repositório.

Estruturas de Repetição

História

Para entender a importância das estruturas de repetição, vamos dar uma volta no passado, mais precisamente no século XIX, e conhecer sua primeira aparição na história. Ada Lovelace, considerada a mãe da programação, era uma jovem matemática que desejava trabalhar ao lado do Charles Babbage no seu projeto da Máquina Analítica - um instrumento capaz de realizar operações matemáticas mais rápido que seres humanos. Após ser recusada inúmeras vezes, Ada escreveu um documento tão detalhado com propostas para uso da Máquina, que Charles aceitou sua colaboração para uma publicação científica.

Suas notas são consideradas o primeiro programa de computador da história!



Máquina Analítica de Charles Babbage

A proposta de Ada que ganhou a história foi uma solução para calcular os números de Bernoulli, uma sequência onde o cálculo de cada elemento depende do elemento anterior. Para facilitar, vamos considerar o somatório dos números inteiros entre 1 e N. Assim como nosso cérebro realiza essa operação, Ada sugeriu **quebrar o problema em um bloco de instruções que resolvem parcialmente o problema, executadas repetidamente até alcançar a solução final**. Essa ideia, na época criativa e inovadora, se tornou a base de uma estrutura fundamental para muitas linguagens de programação modernas, os laços de repetição.

Observe na prática a solução do problema do somatório para 4 números. Do lado esquerdo temos as instruções para solucionar o problema e do lado direito os resultados parciais da solução a cada etapa. Em azul temos o resultado parcial da operação, e em vermelho o ponto da sequência que estamos a cada passo da solução.

1. Inicie o resultado parcial com um valor nulo (zero) 2. Pegue o primeiro valor da sequência **3. Some o resultado parcial com o valor da sequência em mãos** **4. Atualize o resultado parcial com o resultado da soma** 5. Pegue o próximo número da sequência 6. Repita as instruções 3, 4 e 5 até o final da sequência	0 + 1 + 2 + 3 + 4 1 + 2 + 3 + 4 3 + 3 + 4 6 + 4 10

Note que a solução baseada em laços, proposta pela Ada, consiste em:

•executar a operação principal (a soma, na instrução 3) e
•atualizar o estado do problema (instruções 4 e 5) para que a próxima execução avance mais um pouco em direção à solução final.

Na estrutura de repetição, os passos de execução são chamados de **iterações**. No Python, o conceito de **iterável** será muito importante quando falarmos de coleções e estruturas de dados. Conheceremos duas estruturas de repetição, o while e o for. Esse tutorial do Real Python recorre a interessantes definições para eles:

•**Iteração indefinida** (while): baseado em **condições de parada**, o bloco de código se repete um número potencialmente indefinido de vezes, até que se atinja a condição;

•**Iteração definida** (for): O número de repetições está explícito na instrução. No Python, isso é associado a uma sequência finita de elementos, recuperados um a um.

Comando while

A palavra *"while"* em inglês significa "enquanto". Isso torna o entendimento da estrutura mais intuitivo, já que ela repete um bloco de código enquanto uma determinada condição for verdadeira. A condição é qualquer expressão cujo resultado é um valor-verdade, seja uma variável booleana, uma expressão relacional, expressão lógica, entre outros.

Primeiro vamos conhecer a estrutura da sintaxe usando while.

```
1. while <exp>:
2.    # comandos repetidos equanto <exp> valer True
3.    # note o recuo no início da linha.
```

Ela segue a mesma estrutura que aprendemos com o if: a palavra reservada, seguida de uma expressão booleana, e por fim o símbolo : (dois pontos). Assim como na estrutura condicional, o bloco de código interno a estruturas de repetição também **devem estar obrigatoriamente indentados**, ou seja, consistentemente recuados (com tabs ou espaços).

O segredo do while está na expressão que controla o laço. Precisamos garantir que:

- o estado do problema foi previamente **inicializado**
- esse estado será **atualizado** no bloco interno do laço

Vejamos um exemplo prático.

script.py	terminal

`n = 4 # inicializando estado do problema`
`while n > 0: # condição de parada`
`  print(n)`
`  n -= 1 # atualizando estado`
`print("Fim")`
	4
	3
	2
	1
	Fim

Note que usamos uma variável de controle que é inicializada antes do laço (n = 4), definimos a condição para execução do laço (n > 0) e a**tualizamos a variável de controle dentro do laço, garantindo que em algum momento a condição será Falsa** (n -= 1).

Vejamos o passo a passo do fluxo do código:

1) n é inicializado com o valor 4. Isso é importante já que o **teste de condição é a primeira instrução do laço**

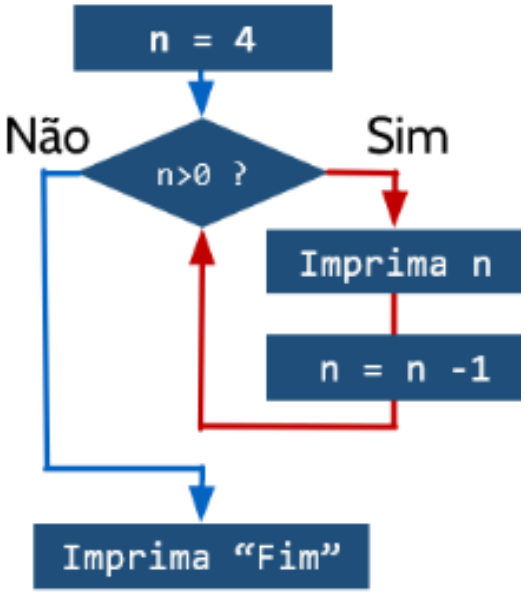
2) A expressão de controle é verdadeira, portanto **entramos no laço**

3) O valor de n é impresso

4) O valor de n é decrementado. Na primeira iteração, n agora vale 3.

5) **Voltamos para o início do laço, testando a condição com o novo valor de n** (instrução 2)

6) Só chegamos aqui quando n é decrementado um número suficiente de vezes para falsear a expressão de controle. Imprimimos "Fim".



Vejamos agora o problema do somatório, solucionado com a estrutura while. Para ficar mais interessante, vamos realizar o somatório dos valores de 1 a N, com N definido pelo usuário.

script.py	terminal
	A soma de 1 a 4 é 10

```
# Entrada de dados
n = int(input("Digite o valor de n"))

# Inicialização de valores
soma = 0  # resultado parcial da soma (nulo)
cont = 1  # contador para controle do laço

# Processamento iterativo (passo a passo)
while cont <= n:  # condição de parada
        soma += cont # operação desejada
        cont += 1    # atualização do controle

# Saída de dados
print(f"A soma de 1 a {n} é {soma}")
```

Para entender passo a passo, vamos isolar o fluxo de execução do laço, acompanhado do estado de todas as variáveis de interesse para solução do problema. Vamos assumir que o usuário entrou com o valor n = 4, no entanto essa solução se aplica a qualquer n.

Iteração	Código comentado	Estado no início → no final

1	`while cont <= n: # 1<=4 ? True`
	`soma += cont #soma += 1`
	`cont += 1 #cont += 1`
		`n = 4 → 4`
		`cont = 1 → 2`
		`soma = 0 → 1`

2	`while cont <= n: # 2<=4 ? True`
	`soma += cont #soma += 1`
	`cont += 1 #cont += 1`
		`n = 4 → 4`
		`cont = 2 → 3`
		`soma = 1 → 3`

3	`while cont <= n: # 3<=4 ? True`
	`soma += cont #soma += 1`
	`cont += 1 #cont += 1`
		`n = 4 → 4`
		`cont = 3 → 4`
		`soma = 3 → 6`

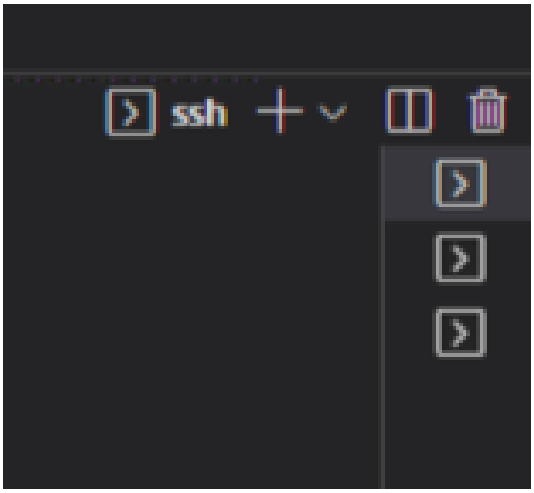
4	`while cont <= n: # 4<=4 ? True`
	`soma += cont #soma += 1`
	`cont += 1 #cont += 1`
		`n = 4 → 4`
		`cont = 4 → 5`
		`soma = 6 → 10`

5	`while cont <= n: # 5<=4 ? False!`	Não há mudança de estado, já que saímos do laço!

Essa solução sempre irá terminar, pois em algum momento a condição será falsa. Caso isso nunca aconteça, teríamos um **loop infinito**. Esse é o erro mais comum de quem está aprendendo a programar, portanto preste atenção (mas não se desespere)! Um loop infinito, como o nome sugere, vai executar infinitamente e você deve forçar a interrupção da execução. Isso pode ser feito das seguintes maneiras:

•No terminal pressione as teclas ctrl+c

•No terminal do VSCode pressione as teclas ctrl+shift+c ou clique no botão com o ícone de lixeira acima do terminal (imagem ao lado).



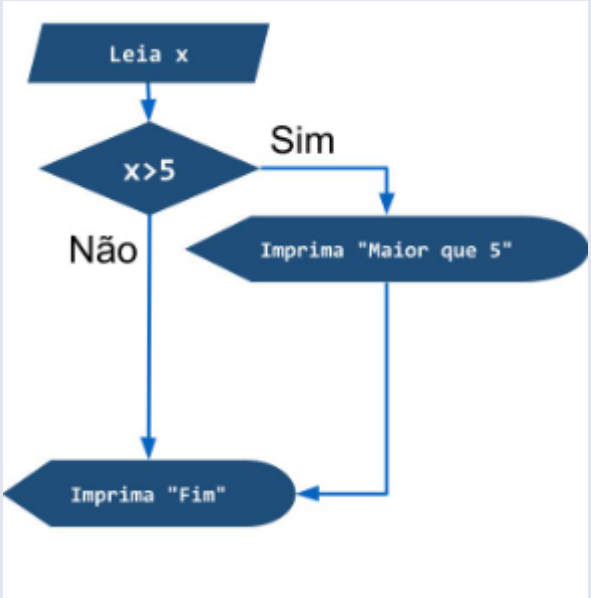
•No Jupyter ou no Google Colab clique no botão com o símbolo de parar, do lado esquerdo da célula em execução, ou vá no menu Ambiente de execução → Interromper execução

Ao aprender os comandos de interrupção break e continue, falaremos mais sobre loops infinitos e casos onde intencionalmente iremos recorrer a eles.

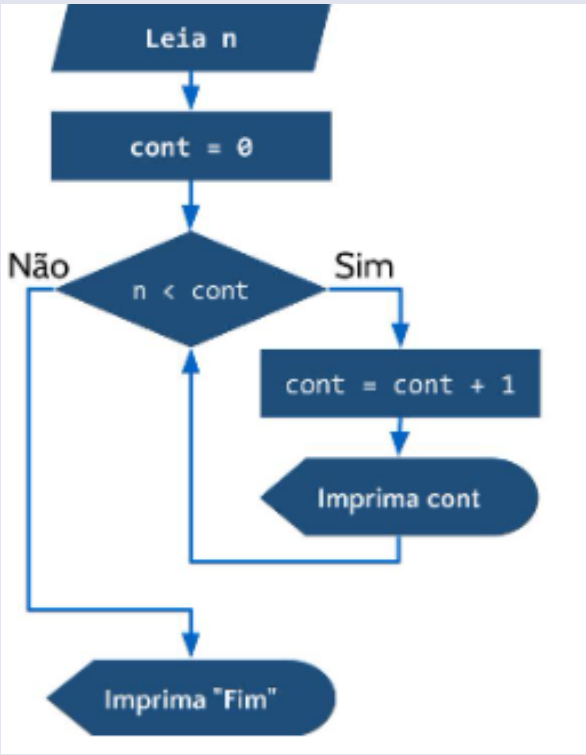
Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula1_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo4 do seu repositório da disciplina no GitHub.

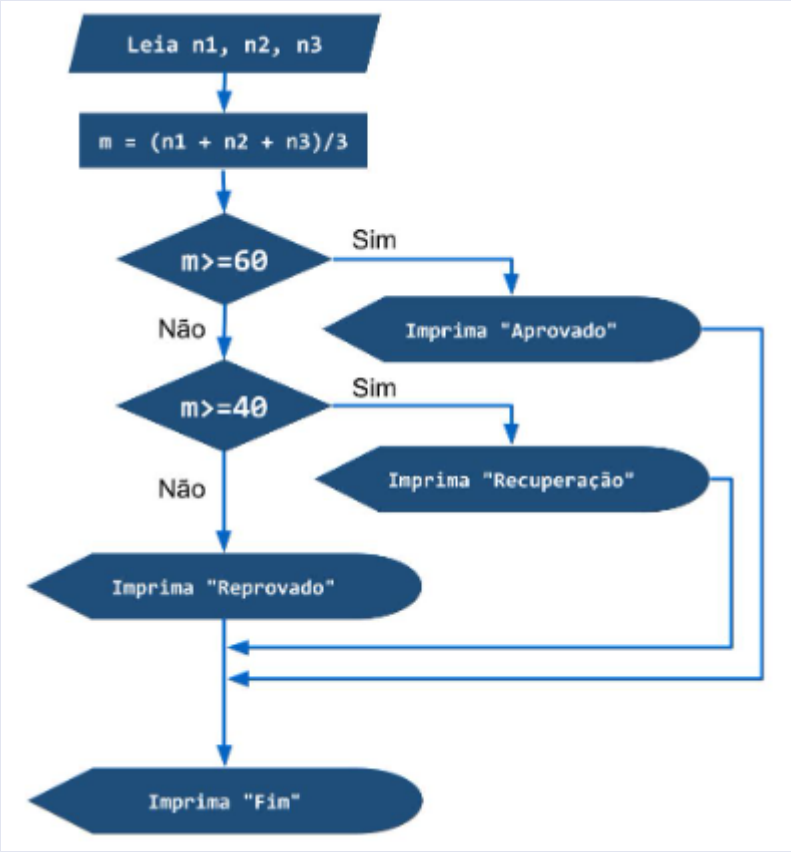
1) Transforme em código o fluxograma a seguir



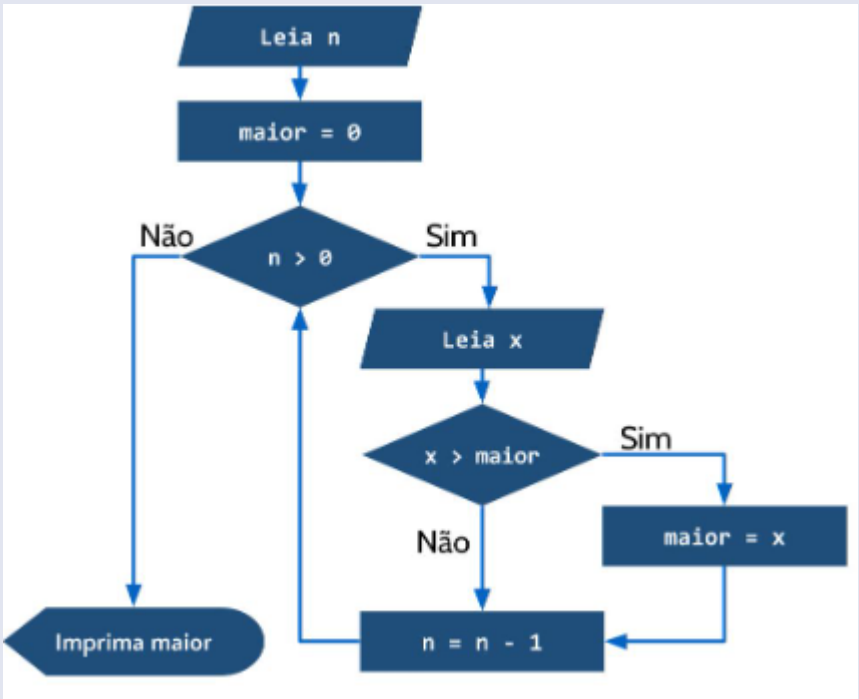
2) Transforme em código o fluxograma a seguir



3) Transforme em código o fluxograma a seguir



4) Transforme em código o fluxograma a seguir



5) Um instituto realizou uma pesquisa de público e precisa calcular a média de idade dos respondentes. Escreva um programa que leia um inteiro N com a quantidade de respondentes e em seguida leia as N idades de cada respondente. Ao final, imprima a média das idades.

(idade1 + idade2 + idade3 + ... + idade_n)/N

6) **Exercício de maratona:** https://www.beecrowd.com.br/judge/pt/problems/view/1094

Maria precisa de sua ajuda para organizar os experimentos de seu laboratório. Ela quer saber no final do ano, quantas cobaias foram utilizadas no laboratório e o percentual de cada tipo de cobaia utilizada. Este laboratório utiliza três tipos de cobaias: sapos, ratos e coelhos.

Entrada: A primeira linha de entrada é um inteiro N com a quantidade de experimentos registrados. As N linhas seguintes contém um inteiro Quantia que representa a quantidade de cobaias utilizadas e um caractere Tipo ('S', 'R' ou 'C') com o tipo de cobaia (S:Sapo, R:Rato ou C:Coelho).

Saída: Apresente o total de cobaias utilizadas, o total de cada tipo de cobaia e o percentual de cada uma em relação ao total de cobaias utilizadas

Exemplo de Entrada	Exemplo de Saída
10 10 C 6 R 15 S 5 C 14 R 9 C 6 R 8 S 5 C 14 R	Total: 92 cobaias Total de coelhos: 29 Total de ratos: 40 Total de sapos: 23 Percentual de coelhos: 31.52 % Percentual de ratos: 43.48 % Percentual de sapos: 25.00 %

Comando for

A palavra *"for"* em português significa "para". A intenção é que a leitura de sua instrução seja algo como ***"para cada elemento de uma dada sequência, ou um dado intervalo, faça algo"***. Sua aparição em diversas linguagens deixa explícito em sua instrução o número de repetições do laço. No entanto, a forma que essa informação é definida pode variar entre linguagens.

Intervalos numéricos: A forma mais simples da estrutura define um intervalo numérico a partir dos valores de início e final, como a primeira instrução do código a seguir, implementado na linguagem BASIC.

Código Basic	Execução
FOR i = 0 TO 3 PRINT i + " " NEXT	0 1 2 3

Expressões de controle: Linguagens como C, C++ ou Java recorrem a instruções compostas por 3 elementos: (1) **inicialização**, executada uma única vez, antes da primeira iteração do laço; (2) expressão booleana servindo de **condição** para execução do laço, assim como vimos no while; e (3) instrução a ser executada ao final de cada iteração, geralmente uma **mudança de estado** na variável de controle do laço.

Código C++	Execução

for (int i = 0; i < 4; i+=1) {
  cout << i << " ";
}
	0 1 2 3

Baseado em sequências: Itera sobre sequências definidas na linguagem, atribuindo a uma variável o próximo elemento da sequência a cada iteração. Existe em inúmeras linguagens, inclusive em algumas das mencionadas anteriormente, e às vezes é conhecido como "foreach" ou "para cada". **É a estrutura nativa do Python.**

Código Python	Execução
for i in range(4): print(i, end=" ")	0 1 2 3

Vamos expandir sobre o terceiro tipo, o for baseado em sequências, já que é a estrutura nativa do Python. Seu objetivo é iterar em um conjunto finito de elementos, com a sintaxe:

```
for <variavel> in <sequência>:
    # comandos executados para cada valor da sequência
```

A cada iteração, <variavel> assume um valor da sequência, com uma condição de parada implícita: o tamanho da sequência. Ainda não falamos de iteráveis e estruturas de dados no Python, mas daremos o primeiro passo conhecendo a função range(), que nos permite replicar o laço baseado em intervalos numéricos, o uso mais fundamental dessa estrutura.

Função range()

Retorna um conjunto finito de valores inteiros, definidos através da seguinte sintaxe:

range([início], fim, [pulo])

Podemos fornecer 3 argumentos para a função, separados por vírgulas, são eles:

- início: Opcional. Valor inicial da sequência [padrão: 0]
- fim: Obrigatório. Final da sequência (não incluso)
- pulo: Opcional. Valor a ser somado a cada novo elemento da sequência [padrão: 1]

Ou seja, obrigatoriamente informamos o último elemento da sequência. Se nada mais for dito, assume-se uma sequência iniciada em 0, incrementando 1 a cada novo elemento, finalizando (mas não incluindo) no elemento informado.

Note no código a seguir que ao chamar a função fora da estrutura de laço, recebemos como retorno um objeto (uma entidade computacional que contém dados e funções).

É preciso iterar no resultado da função range() para acessar os elementos individuais. Entenderemos melhor esse raciocínio quando falarmos de iteráveis no Python.

```
>>> range(5)
range(0, 5)
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Para entender os diferentes usos da função range, vamos apresentar chamadas da função com diferentes parâmetros, e suas equivalências na já conhecida estrutura com while.

Laço for e função range()	Laço while equivalente	Execução
for c in range(5): print(c)	c = 0 while c < 5: print(c) c += 1	0 1 2 3 4
for c in range(1,5): print(c)	c = 1 while c < 5: print(c) c += 1	1 2 3 4
for c in range(1,5,2): print(c)	c = 1 while c < 5: print(c) c += 2	1 3

Os argumentos são interpretados de acordo com a sua posição. Dado 1 único argumento, este sempre será o **fim** da sequência. Dados 2 argumentos, estes serão respectivamente o **início** e o **fim**. Com 3 argumentos, os valores assumem a ordem **início, fim, pulo**. Um comportamento interessante é o pulo negativo, produzindo sequências decrescentes.

Laço for e função range()	Laço while equivalente	Execução
for c in range(4,0,-1): print(c)	c = 4 while c > 0: print(c) c -= 1	4 3 2 1

for c in range(4,-4, -2): print(c)	c = 4 while c > -4: print(c) c -= 2	4 2 0 -2

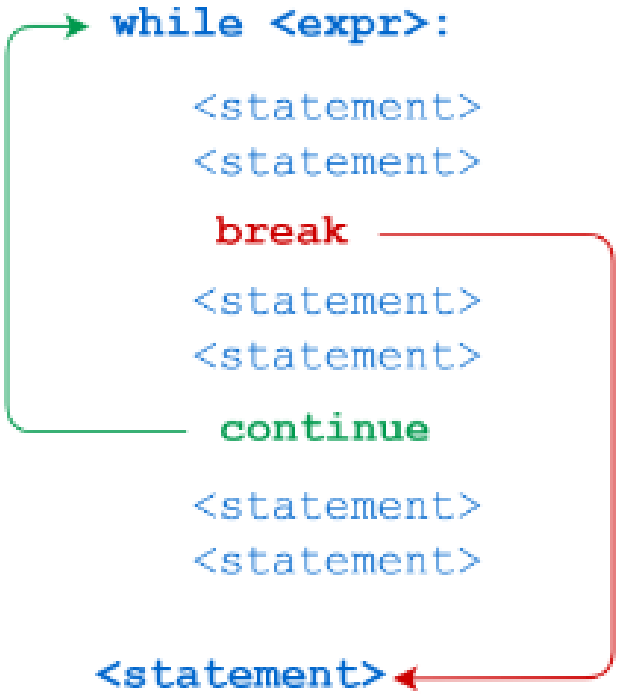
Se sabemos com antecedência o intervalo desejado para o laço, usar o for nos economiza umas boas linhas, além de ficar mais legível. Vamos analisar a solução para o problema do somatório de todos os inteiros entre 1 e N usando for e while.

Somatório com for	Somatório com while	Execução
n = int(input()) soma = 0 for i in range(1, n+1): soma += i print(soma)	n = int(input()) i = 1 soma = 0 while i < n+1: soma += i i += 1 print(soma)	4 3 2 1

Um código que executa de forma muito parecida com ambas as estruturas, nos poupa o controle explícito da variável do laço quando usamos o for. Algo notável também é que usamos n+1 na condição de controle de ambos o range() e a expressão do while, já que ambas testam a condição de forma não inclusiva.

Break e continue

Agora que já estamos ganhando prática com desvios de fluxo no código, vamos ganhar um pouco mais de controle sobre a interrupção de um laço com os comandos break e continue.



A figura ao lado, retirada desse tutorial do Real Python, ilustra os desvios de fluxo de cada comando, cujos comportamentos podem ser descritos como:

break: encerra o laço de repetição, pulando para a próxima instrução depois do laço.

continue: interrompe a execução da iteração atual, não executando os comandos seguintes a ele, voltando para o início do laço e reavaliando a expressão booleana.

É comum utilizá-los quando uma das condições de parada (sim, pode haver várias!) é provocada por uma mudança de estado no meio do corpo do laço. Vamos assumir um jogo de adivinhação, onde uma pessoa tem 5 chances de acertar a fruta do dia. Vejamos o código.

script.py	terminal

fruta = "banana"
for i in range(5):
  palpite = input()
  if palpite == fruta:
  print(">Você acertou!")
  break
print(">Fim de jogo!")
	caju
	abacaxi
	banana
	>Você acertou!
	>Fim de jogo!

Aqui temos duas condições de parada, o intervalo numérico definido por range(5) e a comparação da variável fruta pré-definida com o palpite fornecido pelo usuário. A segunda condição só pode ser testada após capturar o input do usuário, por isso o break serve como uma luva!

Já o continue tem uma intenção diferente, pular a execução do laço (ou parte dele) apenas em alguns cenários. Suponha que iremos ler idade e peso de N pessoas e determinar a média de peso apenas das pessoas acima de 30 anos. Vamos usar o while dessa vez. Quando o teste de condição destacado for verdadeiro, e consequentemente o continue executar, todos os comandos internos do laço após aquele ponto serão pulados e uma nova iteração terá início.

```
N = int(input("Informe a quantidade de pessoas: "))
soma = 0
cont_30 = 0  # um contador para pessoas 30+ (para a média)

while N > 0:
        N -= 1
        peso = float(input(f"Informe o peso da pessoa {i}: "))
        idade = int(input(f"Informe a idade da pessoa {i}: "))
        if idade <= 30: continue
        soma += peso
        cont_30 += 1
print("Média de peso das pessoas acima de 30: ", soma/cont_30)
```

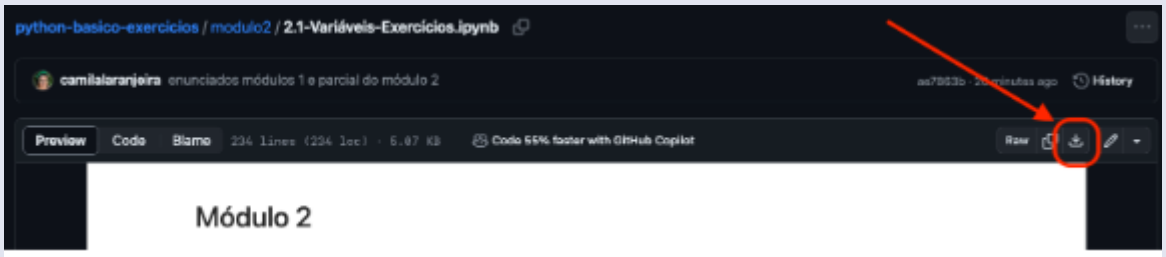
A título de comparação, vejamos a execução de dois códigos muito parecidos, alterando somente o comando de parada. Enquanto o break interrompe completamente o laço, o continue apenas interrompe a iteração atual.

script.py	terminal

print("Exemplo break")
for in range(4):
  if i == 2: break
  print(i)
 	
print("Exemplo continue")
for i in range(4):
  if i == 2: continue
  print(i)
	Exemplo break
	0
	1
	Exemplo continue
	0
	1
	3
	4

Exercício

Baixe o arquivo 4.2-For_Break_Continue-Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.



Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo4 do seu repositório.

Escopo e Funções

Escopo e Funções

Assim como na matemática, funções são estruturas que requerem parâmetros de entrada e produzem um valor de saída. Considere uma função quadrática: f(x) = **a**x^2**+b**x+**c**. Temos aqui uma função de nome f que requer uma **parâmetro** x, a partir desse parâmetro executa uma série de instruções e **retorna** um resultado.

Vamos conhecer nesse módulo 3 tipos de função:

- Funções embutidas: nativas da linguagem;
- Funções de biblioteca: incluídas através de recursos externos;
- Funções customizadas: implementadas no corpo do nosso código.

Funções embutidas e de biblioteca

Já conhecemos algumas funções básicas:

```
x = int(input("Valor: "))
print("O valor de x é", x)
```

Os comandos int, input, print, que já utilizamos em módulos anteriores, são funções nativas do Python. Em outras palavras, podemos simplesmente chamá-las em nosso código sem incluir ou implementar nenhum recurso extra. A lista completa de funções embutidas está na documentação do Python, mas vamos conhecer uma boa parte delas até o final deste curso.

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Já sabemos que as funções:

•**podem ter** entrada e saída (mas nem sempre tem);

•usam a sintaxe dos parênteses para receber parâmetros de entrada;

•podem receber múltiplos parâmetros separados por vírgulas;

•são expressões simples que podem dar retorno, o qual podemos armazenar, imprimir, etc., assim como qualquer expressão.

retorno = nome_da_funcao(conjunto, de, parametros)

O que ainda não comentamos é que existem diversos **módulos** e **pacotes** que aumentam a quantidade de funções, entre outros recursos, ao nosso dispor. Esses recursos adicionais podem ser importados, ou incorporados, aos nossos scripts. Antes de seguir, vamos definir cada um desses termos:

•**Script:** um único arquivo de código que será executado para servir algum propósito;
•**Módulo:** um único arquivo cujo objetivo é ser importado em outros scripts ou módulos;
•**Pacote:** conjunto de módulos estruturados em uma única pasta, que podem ser importados como uma entidade única em scripts e módulos.

Temos também um termo mais abstrato, **biblioteca**, definido como um **conjunto de recursos que podemos importar em nosso código.** É comum se referir a módulos como "bibliotecas de funções", assim como pacotes que contém centenas ou milhares de módulos também se definem como bibliotecas.

Quando instalamos o Python, obtemos não só o interpretador de código, como também a "Python Standard Library" ou "biblioteca padrão do Python". Ela traz consigo centenas de módulos que reúnem recursos de mesma natureza, como o math, de recursos matemáticos, ou o random, para gerações de valores aleatórios e distribuições das mais variadas.

•math: log(), cos(), atan(), sqrt(), pi, inf ...
•random: random(), randint(), uniform(), gauss(), ...

Para usar funções de algum módulo da biblioteca padrão, basta importar o módulo desejado com a palavra reservada import. Em geral cria-se um cabeçalho no início do código com todos os módulos e pacotes necessários. No exemplo a seguir, importamos o módulo random, utilizando sua função randint dentro de um laço para gerar 4 valores aleatórios no intervalo entre 0 e 100. Pela natureza da função, é provável que você obtenha um resultado distinto ao executar o código. Note que a chamada da função segue a sintaxe biblioteca.funcao().

```
import random
for i in range(4):
    print(random.randint(0, 100), end=" ")
```

```
41 82 1 8
```

Bibliotecas também podem ter variáveis imutáveis (constantes), como no caso da math que traz constantes como o valor de pi, uma representação do infinito (inf), entre outros.

```
import math
raio = float( input("Digite o raio da circunferência: ") )
print(f"Sua área é {math.pi * raio**2}")
```

Consulte a documentação para mais informações sobre a biblioteca padrão. Conheceremos alguns de seus módulos ao longo do curso.

Como descrito na documentação, Python tem uma comunidade muito ativa de pessoas que tornam seus softwares disponíveis publicamente através de repositórios como o pypi.org. Através de um gerenciador de pacotes, podemos instalar (ou desinstalar, atualizar, etc.) outros recursos além da biblioteca padrão. No Python, temos o pip, cujo nome tem dois significados populares na comunidade, "preferred installer program" ou "programa de instalação preferido", além do acrônimo recursivo (uma sigla onde uma das letras é a própria sigla) "pip installs packages" ou "pip instala pacotes".

As versões mais recentes do Python já trazem o pip consigo, no entanto, para nos certificarmos de sua instalação, execute o seguinte comando no terminal:

```
$ python -m ensurepip --default-pip
```

Em seguida, cheque a versão instalada (pra garantir que tá tudo certo!)

```
$ pip --version
```

Por fim, para instalar um pacote use o seguinte comando

```
$ pip install nome_do_pacote
```

Se você deseja especificar a versão que deseja instalar (faremos muito isso!) é só adicionar a informação da seguinte maneira:

```
$ python -m pip install nome_do_pacote==1.0.4     # specific version
$ python -m pip install "nome_do_pacote>=1.0.4"  # minimum version
```

Consulte a documentação para saber mais sobre a instalação de bibliotecas no Python.

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula1_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo5 do seu repositório da disciplina no GitHub.

1) Desenvolva um programa em Python que peça ao usuário para inserir dois números decimais e calcule a diferença absoluta entre esses dois números. Utilize a função nativa abs para garantir que o resultado seja sempre positivo e round para arredondar o resultado para duas casas decimais.

Exemplo de interação:

```
Digite o primeiro número: 3.1415
Digite o segundo número: 1.4142
A diferença absoluta entre os números é: 1.73
```

2) Escreva um código que gere n valores inteiros aleatórios entre 0 e 100 e calcule a raíz quadrada da soma dos valores. Peça ao usuário o valor de n

•Biblioteca random: Função randint()
•Biblioteca math: Função sqrt()

3) Escreva um programa em Python que utiliza a biblioteca random para gerar um número aleatório entre 1 e 10 e peça ao usuário para adivinhar o número. Forneça feedback se o palpite é muito alto, muito baixo ou correto. Interrompa a execução somente quando o usuário acertar o palpite.

Exemplo de interação:

```
Adivinhe o número entre 1 e 10: 5
Muito alto, tente novamente!
Adivinhe o número entre 1 e 10: 3
Correto! O número é 3.
```

4) Escreva um programa em Python que utiliza a biblioteca datetime para exibir a data e hora atuais com a formatação apresentada a seguir:

```
Data: 15/03/2023
Hora: 14:05
```

Você pode consultar esse tutorial da Alura sobre a biblioteca datetime. Existem várias maneiras de resolver essa questão. Uma sugestão é:

•Função datetime.datetime.now() cujo retorno possui os atributos: day, month, year, hour, minute
•Usar a formatação com f-strings no momento de imprimir. Atenção para os atributos que devem sempre ter 2 dígitos precedidos por zero se necessário.

5) Você está trabalhando em um sistema de mensagens instantâneas, que deve permitir o uso de emojis nas conversas entre pessoas. Faça:

•No terminal, instale a biblioteca emoji usando o gerenciador de pacotes pip

```
$ pip install emoji
```

• Conheça a função emoji.emojize()

Seu programa deve apresentar para o usuário a lista de emojis disponíveis com o texto correspondente a cada emoji. Em seguida, solicite uma frase codificada ao usuário e apresente ela decodificada com a visualização de emojis (emojizada).

A seguir um exemplo de interação, com uma lista de emojis sugeridos. Você pode consultar o texto que codifica outros emojis indo nessa página e passando o mouse por cima do emoji desejado.

```
Emojis disponíveis:
❤️ - :red_heart:
👍 - :thumbs_up:
🤔 - :thinking_face:
🥳 - :partying_face:

Digite uma frase e ela será emojizada:
Olá mundo! :red_heart:
Frase emojizada:
Olá mundo! ❤️
```

Funções Customizadas

Funções são blocos de código nomeados que podem ser chamados inúmeras vezes para executar seu conteúdo. Assim como criamos variáveis, damos nomes a elas e associamos algum conteúdo, criamos funções com nomes (relativos à sua tarefa) e associamos um bloco de código. Esses blocos podem ser reutilizados quantas vezes for necessário apenas chamando-os pelo nome da função. Ótimo pra reusabilidade! Podemos executar o mesmo bloco sem reescrevê-lo, e até com valores distintos a cada execução. Além de ser melhor para a organização do nosso código. Reunir um conjunto de instruções em um bloco separado melhora a leitura do código principal.

Vejamos a seguir a estrutura de sintaxe para criar uma função. Assim como comandos de decisão e de repetição, as instruções internas à função têm um espaçamento no início da linha, uma indentação obrigatória. O programa principal retorna ao recuo original.

```
def nome_da_funcao(p1, p2, p3):
    valor = 0
    # instruções internas da função...
    return valor
# programa principal
```

Palavra reservada que indica definição de função

Valores de entrada (parâmetros)

Palavra reservada que precede os valores de saída da função

Vamos entender um a um os elementos obrigatórios de uma função. Quando aprendemos os elementos de uma solução algorítmica, falamos de **entrada**, **processamento** e **saída**. Aqui vale a mesma lógica, mas no contexto de funções usaremos uma nova nomenclatura: **parâmetros**, **corpo da função** e **retorno**.

•**Assinatura:** Primeira linha da função. Deve obrigatoriamente ter a palavra reservada def, seguido do nome da função, um par de parênteses (vazios ou não), finalizando com o símbolo : (dois pontos). Dentro dos parênteses são definidos os **parâmetros**, variáveis de entrada da função, cujos valores serão fornecidos quando a função for chamada. A existência de parâmetros é opcional (nem toda solução precisa de entrada).

•**Corpo:** Contém as instruções que serão executadas **somente quando a função for chamada.** Ao ser definida, as instruções internas da função não serão executadas, apenas associadas ao nome que a define.

•**Retorno:** Ao final, a função pode ou não retornar um valor resultado, o qual deve ser precedido da palavra reservada return. Neste ponto a função é finalizada e o fluxo de execução retorna para a linha onde ela foi chamada.

Vejamos um exemplo de função acompanhado por setas que indicam os desvios de fluxo.

```
import math
def distancia(x, y):
    d = math.sqrt(x**2 + y**2)
    print(d)

print("Início")
distancia(3, 1)
print("Fim")
```

A função ao lado, chamada distancia, tem dois parâmetros de entrada, x e y, e nenhum retorno. Sua execução é da seguinte forma: (1) importação do módulo math; (2) definição (e não execução) da função; e (3) programa principal, imprimindo a palavra "Início" e então **desviando o fluxo** para executar o bloco de código da função. Após a última instrução da função, o fluxo retorna para o próximo comando depois da chamada da função.

Para executar a função, basta criar uma instrução chamando-a pelo seu nome, obedecendo os critérios de entrada e saída. A seguir temos dois exemplos de função, uma com dois parâmetros de entrada e uma saída e outra com 3 parâmetros de entrada e nenhuma saída.

def subtracao(a, b):
  return a - b
 	
# programa principal 	
s = subtracao(2, 3)
print(s)
	def subtracao(a, b, c):
	  print(a - b - c)
	
	# programa principal
	subtracao(2, 3, 5)

-1	-6

Intuitivamente, os valores de entrada fornecidos na chamada da função são associados aos parâmetros pela posição. Na função à esquerda:

•Os valores dos parâmetros são **a=2** e **b=3**

•O valor de retorno (resultado da expressão **a-b**) é armazenado na variável **s** do programa principal.

Alguns parâmetros podem ser **opcionais**, ou seja, não fornecidos na chamada da função, desde que acompanhados de um valor padrão. No exemplo a seguir, imprime será verdadeiro a menos que se explicite o contrário.

def sub(**a, b, imprime**=**True**):
  if (imprime): print(a-b)
  else: **return** a - b
 	
# programa principal 	
sub(**2, 3**)
print("Fim")
	def sub(**a, b, imprime**=**True**):
	  if (imprime): print(a-b)
	  else: **return** a - b
	
	# programa principal
	s = sub(**2, 3, False**)
	print("Fim")

-1 Fim	Fim

Podemos explicitar o nome dos parâmetros no momento da chamada da função, isso nos permite desobedecer a ordem de envio dos parâmetros. Essa é a diferença entre passagem de parâmetro **posicional** e parâmetro **nomeada** (por palavra-chave). No entanto, uma vez que nomeamos um parâmetro na chamada, precisamos nomear todos os seguintes já que a ordem foi desrespeitada a partir daquele ponto. Se esquecermos de nomear, temos um erro de sintaxe que diz *"argumento posicional segue argumento por palavra-chave"*. Argumento é outra palavra para parâmetro.

```
def sub(a, b, imprime=True):
        if (imprime): print(a-b)
        else: return a - b

# programa principal
s = sub(imprime=False, b=2, a=3)
print(s)
```

```
-1
```

```
def sub(a, b, imprime=True):
        if (imprime): print(a-b)
        else: return a - b

# programa principal
s = sub(imprime=False, 2, 3)
print(s)
```

```
File "<ipython-input-1-419ed4e-4d99e>", line 6
        s = sub(imprime=False, 2, 3)
                                ^
SyntaxError: positional argument follows keyword argument
```

Em geral, passamos os parâmetros obrigatórios (que não possuem valor padrão) por posição. Parâmetros opcionais são passados apenas quando necessário, sua ordem importa menos.

```
def subtracao(a, b, imprime=True, absoluto=False):
        if (imprime): print(a-b)
        else: return abs(a - b) if absoluto else a-b

# programa principal
subtracao(2, 3, absoluto=True)
```

Da mesma forma que se pode ter múltiplos parâmetros na mesma função, também é possível ter múltiplos retornos, basta separá-los por vírgulas.

```
import math
def cart2polar(x, y):
        dist   = math.sqrt(x**2 + y**2)
        angulo = math.atan2(y, x)
        return dist, angulo

d_polar, a_polar = cart2polar(3, 1)
```

Documentando suas funções

Por último, mas não menos importante, quando escrevemos funções, criamos "caixas-pretas" que você ou colegas vão chamar em seus programas. É recomendável documentar suas funções com comentários logo acima dela, indicando seus principais elementos: entrada, funcionamento e saída. Vamos usar muito a documentação para entender funções novas, então documente também as suas funções! Acredite, nossa memória é curta mesmo para nossos próprios códigos.

No código a seguir temos um exemplo simples de estrutura de comentários, e no futuro você conhecerá ferramentas de documentação capazes de ler esses comentários e produzir documentos completos descrevendo nossos códigos. Então crie o hábito desde já :)

```
# Aplica a operação de subtração
# a duas variáveis.
#
# Params:
#   a: variável numérica
#   b: variável numérica
# Retorno:
#   s: variável numérica
def subtracao(a, b):
    s = a - b
    return s
```

Escopo

No português, a palavra "escopo" significa "limite ou espaço de ação". Quando dizemos por exemplo que uma tarefa está "fora do escopo", significa que ela não pode ser realizada em um dado contexto. Na programação, elementos nomeados, como variáveis e funções, têm escopo definido e não podem ser acessados em trechos de código fora desse escopo.

O primeiro exemplo prático disso é o início do escopo de uma variável, que só passa a existir a partir do momento que é declarada. No exemplo a seguir, condicionamos a criação da variável c ao valor de a. Se a condição não é verdadeira, a variável c não será criada e, portanto, seu escopo não será iniciado. Uma vez criada no programa principal, o escopo da variável se estende até o fim do programa (a menos que seja deletada).

a = 10
if a > 0:
  c = 1
print(c)
	a = -1
	if a > 0:
	  c = 1
	print(c)

1	Traceback (most recent call last): File "<stdin>", line 1, in <module> NameError: name 'c' is not defined

Ao declarar uma função, criamos um escopo separado do programa principal. Em outras palavras, variáveis criadas dentro de uma função deixam de existir ao final da chamada. No código a seguir, tentamos acessar uma variável local, chamada valor, fora do seu escopo. Por isso, obtemos o mesmo erro do código anterior informando que o nome acessado não está definido (naquele escopo).

```
def minhaFuncao(parametro):
    valor = parametro * 2
    return valor

dobro = minhaFuncao(0.5)
print(valor)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'valor' is not defined
```

Ou seja, o escopo local de funções não é acessível a partir do programa principal. Parâmetros e retornos servem justamente para que esses escopos separados possam se comunicar. Por outro lado, variáveis declaradas no programa principal são consideradas **globais** (acessíveis em qualquer trecho do código após sua criação). Por isso conseguimos imprimir o valor de uma variável do programa principal dentro do escopo de uma função.

```
def func():
    print("Na função: ", x)

x = 10
func()
print("No programa principal: ", x)
```

```
Na função: 10
No programa principal: 10
```

Porém, cuidado! Ao atribuir valor pela primeira vez a uma variável dentro de uma função, a instrução é tratada como a criação de uma nova variável local, mesmo que se use o mesmo nome de uma variável do programa principal. Apesar de confuso, esse comportamento cria o saudável hábito de usar a passagem de parâmetros para acessar variáveis do programa principal. No código a seguir, existem

duas variáveis distintas com o mesmo nome, uma local e uma global, isso só é possível por pertencerem a escopos separados. Note que ao retornar da chamada da função, o valor do x global (programa principal) permanece inalterado.

```
def func():
    x = 1  # x local
    print("Na função:", x)

x = 10     # x global
func()
print("No codigo principal:", x)
```

```
Na função: 1
No programa principal: 10
```

Caso deseje alterar uma variável global em um escopo local, é possível usar a palavra reservada **global** para explicitar esse uso. Com uma única linha de código explicitamos que atribuições de valor à variável x se referem ao x do programa principal.

```
def func():
    global x
    x = 1  # x global
    print("Na função:", x)

x = 10     # x global
func()
print("No codigo principal:", x)
```

```
Na função: 1
No programa principal: 1
```

Funções Lambda

No Python, assim como na maioria das linguagens, temos o conceito de funções anônimas de sintaxe concisa (uma única expressão). Você pode ler nesse tutorial do Real Python sobre o surgimento desse conceito, que tem suas raízes no Cálcu-

lo Lambda. Aqui nos concentramos na estrutura sintática e funcionamento básico de funções anônimas do Python, o qual será expandido em módulos futuros.

As chamadas funções lambda suportam qualquer expressão, porém não armazenam estado. Sua sintaxe segue a estrutura:

```
lambda <parametros>: <expressão>
```

Em termos práticos, é uma função capaz de receber valores de entrada (assim como todas as outras funções), porém o corpo da função é definido por uma única expressão que opera nos parâmetros de entrada. O resultado dessa expressão será retornado ao final.

Assim como outras funções, ao escrevermos uma função lambda, não estamos executando a expressão, apenas definindo a estrutura. Mas diferente de outras funções, esta é uma **expressão cujo retorno é a própria função**. Se executarmos no terminal interativo apenas a expressão lambda, vemos no retorno dado uma representação do objeto **função** <lambda>. O retorno **pode ser atribuído a uma variável** para uso futuro, ou seja, para aplicar a função.

```
>>> lambda x: x + 1        # apenas definindo, retorna a própria função
<function <lambda> at 0x102d913a0>
>>> func = lambda x: x + 1 # atribuindo a função a uma variável
>>> func(3)                # invocando a execução da função
4
```

A função lambda acima equivale (em parte) a **def func(x): return x + 1**

Porém, diferente de outras funções, esta é **anônima**, portanto não temos obrigação de nomeá-la. Usando a sintaxe de parênteses apresentada a seguir podemos simultaneamente definir e executar uma função lambda.

```
>>> (lambda x: x + 1)(3)  # definindo e executando a função
4
```

Pela sua natureza anônima, o interpretador vai se referir a ela como <lambda> mesmo se associadas a uma variável, enquanto funções definidas com def são referidas pelo nome. Precisamos prestar atenção nesse comportamento para conseguir rastrear nossos erros.

```
>>> def func_definida(x): return x + 1
'''
>>> func_definida('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in func_definida
TypeError: can only concatenate str (not "int") to str

>>> func_anonima = lambda x: x + 1
>>> func_anonima('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
TypeError: can only concatenate str (not "int") to str
```

Dar suporte a uma única expressão pode parecer uma grande desvantagem, mas lembre-se que Python é uma linguagem muito versátil, trazendo até estruturas como as condicionais na forma de expressões.

```
fn_maior = lambda a, b: a if a>b else b
for i in range(10):
   x, y = int(input()), int((input))
   print(fn_maior(x, y))
```

Mas atenção! Suportar uma única expressão não é sinônimo de uma única linha. Podemos envolver a expressão em parênteses ou usar o caractere "\" ao final da linha para escrever expressões únicas ao longo de múltiplas linhas.

func = (lambda x: print("x>0") if x > 0 else print("x<=0"))	func = lambda x: print("x>0") \ if x > 0 else \ print("x<=0")

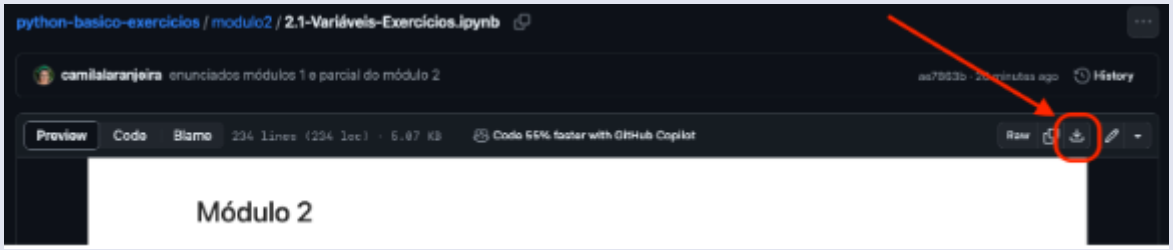
Assim como funções definidas, funções lambda também aceitam parâmetros obrigatórios ou opcionais, posicionais ou nomeados.

```
>>> (lambda x, y, z: x + y + z)(1, 2, 3)
6
>>> (lambda x, y, z=3: x + y + z)(1, 2)
6
>>> (lambda x, y, z=3: x + y + z)(1, y=2)
6
```

A princípio vamos usar expressões lambda para escrever funções menores de maneira mais concisa, porém existem muitos usos associados a estruturas de dados, recursividade, entre outros que ainda não vimos.

Exercício

Baixe o arquivo 5.2-Funções_Customizadas_Lambda-Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.

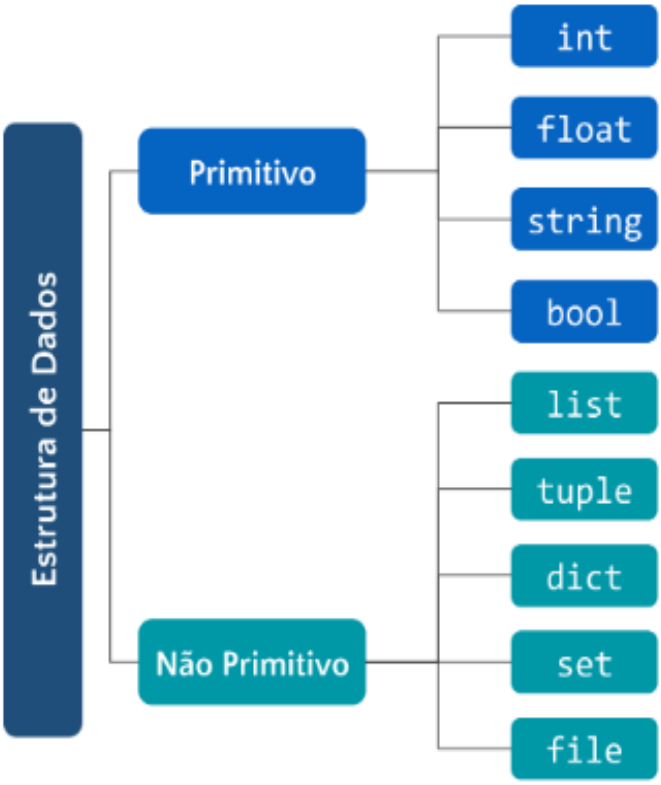


Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo5 do seu repositório.

Listas

Listas

Até agora conhecemos tipos primitivos do Python: **int, float, string, bool**. Ainda vamos conhecer as nuances do tipo **string**, que o distingue dos demais mas, de forma geral, tipos primitivos são simples e atômicos, ou seja, comportam um único valor. Existem tipos mais complexos nativos da linguagem que permitem criar coleções de dados primitivos. São as **estruturas de dados**, que no Python recebem o nome de **containers**. Elas permitem organizar e manipular coleções de dados primitivos. São elas:



- **Lista**
- Tupla
- Dicionário
- Set

Propriedades de listas

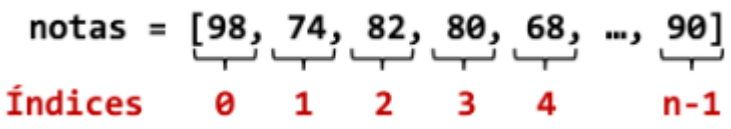
A lista é o container mais utilizado do Python, pela sua versatilidade e flexibilidade de uso. Mas antes de entrar nas especificidades desta estrutura, vamos entender a sua importância. Imagine que em um dado problema precisamos armazenar as notas de 3 alunos em uma prova. Com o que sabemos até agora, basta criar 3 variáveis e o problema está resolvido.

```
nota1 = int(input("Digite a primeira nota"))
nota2 = int(input("Digite a segunda nota"))
nota3 = int(input("Digite a terceira nota"))
```

Mas o que acontece se precisamos armazenar 100 notas? Ou pior, se falarmos de um sistema como o do ENEM, que em 2023 teve quase 4 milhões de candidatos! É inconcebível criar milhões de variáveis independentes, o que tornaria inviável extrair métricas simples como a maior nota, média das notas, entre outros. É aí que entram os containers!

Com a lista, podemos armazenar múltiplos valores na mesma variável de forma **ordenada**. Na prática, isso significa que cada elemento possui um índice referente à sua posição na estrutura, posição essa que é definida pela ordem que os elementos foram adicionados. Na imagem a seguir, podemos ver tanto a sintaxe de criação de uma lista, quanto o padrão de indexação (organização dos índices) que varia de 0 a n-1, sendo n o número de elementos. Sua sintaxe de definição consiste em separar os valores por vírgula dentro de colchetes.

Para acessar um elemento da lista, basta indicar o índice que se deseja acessar, colocando-o dentro de colchetes após o nome da lista.



```
>>> notas = [98, 74, 82, 80, 68, 90]
>>> notas[2]
82
>>> notas[0]
98
>>> notas[5]
90
```

Assim como aprendemos com a função range que valores negativos indicam um caminhamento de trás para frente, nas listas a lógica é a mesma. Índices negativos permitem acessar a lista de trás pra frente. Neste caso os índices variam de **-1** a **-n**, respectivamente o último e o primeiro elemento.

```
>>> notas[-1]
90
>>> notas[-6]
98
```

A segunda propriedade relevante sobre listas é ser **mutável**, ou seja, suporta alterações após a criação da lista. Essa é uma das propriedades que tornam a lista tão flexível, mas futuramente conheceremos estruturas imutáveis para entender que há vantagens nisso.

notas = [98, 74, 82, 80, 68, 80] **print(notas)** **notas[2] = 100** **print(notas)**	**[98, 74, 82, 80, 68, 80]** **[98, 74, 100, 80, 68, 80]**

Até então vimos a lista dentro do escopo do problema de armazenar notas, onde todos os elementos são do mesmo tipo. No entanto, listas são ainda mais poderosas, pois são uma estrutura **heterogênea**, ou seja, suporta tipos diferentes de valores na mesma variável, inclusive outras listas.

lista = ["verde", 28, 94.6, True] **print(lista)** **lista[3] = [1,2,3]** **print(lista)** **print(lista[3], lista[3][0])**	**["verde", 28, 94.6, True]** **["verde", 28, 94.6, [1,2,3]]** **[1,2,3] 1**

Apesar de ainda não conhecermos o conceito de classes e objetos, já comentamos que tipos de dados no Python são objetos, e portanto possuem dados e funções. A função embutida dir() nos permite consultar as **funções membro** de um determinado objeto.

```
>>> lista = ["verde", 28, 94.6, True]
>>> dir(lista)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', ..., '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

No exemplo acima, destacamos em laranja as funções que podem ser chamadas através da sintaxe **objeto.funcao()**, assim como vimos em funções de bibliotecas. Todas as outras, que seguem o formato **__func__**, são chamadas de funções mágicas, ou **métodos mágicos**. Estes são invocados pelo Python em determinadas operações, como funções embutidas (como o próprio dir), operações aritméticas (soma, subtração, etc.), entre outros. Em disciplinas futuras iremos nos aprofundar nestes métodos, mas você pode conhecê-los nesse tutorial do Real Python ou na documentação oficial.

Para finalizar a nossa introdução às listas, precisamos conhecer a função **append()**. Ela adiciona novos elementos no final da lista, mantendo a ordem dos elementos já existentes. Em português, a palavra significa "acrescentar" ou "anexar". Isso significa que **a lista é uma estrutura que pode crescer ou diminuir dinamicamente**. Note no exemplo a seguir que a função append não possui retorno, apenas alterando a estrutura interna da lista.

```
>>> notas = [68, 75, 70]
>>> notas.append(100)
>>> print(notas)
[68, 75, 70, 100]
```

No exemplo a seguir temos um uso comum de listas, onde inicializamos uma lista vazia e em seguida usamos a função para preenchê-la iterativamente. Para criar uma lista vazia podemos usar tanto a sintaxe de colchetes vazios, quanto invocar **list()** sem parâmetros.

```
>>> lista = []
>>> for i in range(10):
...     lista.append(i)
...
>>>> lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Começamos a conhecer as propriedades fundamentais das listas. Em resumo:

•**Ordenada:** Elementos organizados de maneira sequencial, por ordem de adição. Nos permite acessar elementos pelo índice (indexar), com índices que variam de **0** a **n-1**, na escala positiva ou **-1** a **-n**, na escala negativa (ordem inversa).

•**Mutável:** Suporta alterações após a sua criação. Seus valores podem não só ser alterados, como a estrutura também cresce e diminui dinamicamente. Veremos nas próximas aulas mais operações de adição e remoção de elementos.

•**Heterogênea:** Capaz de armazenar variáveis de diferentes tipos na mesma estrutura, inclusive listas dentro de listas (estrutura aninhável).

Até o final deste módulo nos aprofundaremos em duas importantes propriedades:

•**Iterável:** Suporta iterações sobre seus elementos
•**Fatiável:** Suporta extração de subconjuntos de seus elementos

Operações e funções de listas

Como dissemos na aula anterior, funções mágicas definem comportamentos para operações nativas do Python. No caso das listas, por exemplo, temos as operações de adição (__add__) e multiplicação (__mul__). Elas definem o comportamento dos operadores + e * quando estes operam sobre listas.

```
>>> dir(list)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dir__', ..., '__len__', '__lt__', '__mul__', '__ne__', ..., 'append',
'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Para listas, a soma é uma operação binária lista1 + lista2, produzindo como saída uma nova lista concatenando os elementos dos seus operandos.

```
notas1 = [68, 75, 70]
notas2 = [77, 89, 80]
notas_concat = notas1 + notas2
print(notas_concat)
notas_concat += [100]
print(notas_concat)
```

```
[68, 75, 70, 77, 89, 80]
[68, 75, 70, 77, 89, 80, 100]
```

Já a multiplicação é uma operação binária lista * inteiro, concatenando instâncias repetidas da mesma lista dada.

```
lista1 = [2]
print(lista1 * 5)
print([1,2,3] * 2)
```

```
[2, 2, 2, 2, 2]
[1, 2, 3, 1, 2, 3]
```

Outra função mágica importante para nós neste momento é a __iter__, caracterizando a lista como um objeto **iterável**. Como o nome sugere, um iterável é um objeto que dá suporte a iterações, ou seja, conseguimos atravessar a coleção elemento a elemento. A função **__iter__** é responsável por retornar um **iterador**, que controla a visita dos elementos retornando um novo elemento a cada iteração.

```
>>> dir(list)
['__add__', '__class__', ..., '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', ..., '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Podemos por exemplo utilizar uma lista na estrutura do for, de modo que os elementos serão visitados pela ordem dos índices (ordem de adição) a cada iteração.

```
notas = [98, 74, 82, 80]
for n in notas:
    print(n, end=" ")
```

```
98 74 82 80
```

No Python existem algumas funções nativas que também operam sobre iteráveis, entre elas:

•**len:** retorna o número de elementos no iterável
•**max:** retorna o elemento de maior valor no iterável
•**min:** retorna o elemento de menor valor no iterável
•**sum:** retorna a soma de todos os elementos no iterável

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums)) #media
25
```

Em alguns casos, precisaremos iterar nos índices de uma coleção. De posse do que já sabemos, uma forma é combinar as funções **len** e **range**.

```
notas = [98, 74, 82, 80]
tamanho = len(notas)
for i in range(tamanho):
    print(f"Elemento {i}: {notas[i]}")
```

```
Elemento 0: 98
Elemento 1: 74
Elemento 2: 82
Elemento 3: 80
```

Vamos conhecer mais algumas funções membro de listas. Caso você queira descrições detalhadas sobre todas as funções disponíveis, consulte a documentação.

```
>>> dir(list)
['__add__', '__class__', ..., 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

•**lista.index(x):** Retorna a primeira ocorrência do valor x na **lista**.
•**lista.count(x):** Retorna o número de ocorrências do valor x na **lista**.
•**lista.sort():** Ordena os valores da lista, **modificando a lista original**. Ou seja, não produz retorno.

```
>>> nums = [3, 42, 12, 9, 3, 74]
>>> lista.index(42)
1
>>> lista.index(3)
0
>>> lista.count(3)
2
>>> lista.sort()
>>> lista
[3, 3, 9, 12, 42, 74]
```

O Python possui outra maneira de ordenar os valores de uma lista, através da função embutida **sorted()**. Vale destacar as diferenças fundamentais entre a função embutida **sorted()** e a função de listas **sort()**.

•**sort()** é uma função membro da estrutura da lista e portanto é invocada pela sintaxe lista.sort(). Também modifica a lista original, sem produzir retorno.
•**sorted()** é uma função embutida do Python que aceita iteráveis como parâmetro, ou seja, **sorted(lista)**. Ela **retorna uma nova lista** com os valores ordenados.

Já mencionamos que listas são dinâmicas e portanto podem aumentar ou diminuir de tamanho após sua inicialização. Assim como a função **append** aumenta a lista, adicionando novos elementos ao final, as funções **pop** e **remove** reduzem seu tamanho.

•**lista.pop(i):** Remove o elemento no índice **i** da **lista**, e **retorna este elemento**.

•**lista.remove(x):** Remove a primeira ocorrência do elemento x na **lista**, e não produz retorno.

```
>>> a = [0, 2, 3, 2]
>>> a.pop(2)
3
>>> a
[0, 2, 2]
>>> a.remove(2)
>>> a
[0, 2]
```

Fora de uma iteração, a expressão x **in lista** testa se o elemento x pertence à **lista**. É uma expressão de retorno booleano, retornando **True** caso o elemento esteja presente e **False** caso contrário.

notas = [98, 74, 82, 80, 68, 90] print(98 in notas) print(-1 in notas) print(80 in notas)	True False True

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula2_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo6 do seu repositório da disciplina no GitHub.

1) Faça um programa que gere aleatoriamente 20 valores inteiros entre -100 e 100 e os armazene em uma lista. Em seguida imprima na ordem estabelecida:

•A lista ordenada, sem modificar a lista original
•A lista original
•O índice do maior valor da lista
•O índice do menor valor da lista

2) Faça um programa que gere aleatoriamente um valor entre 5 e 20 e armazene em uma variável chamada num_elementos. Em seguida gere aleatoriamente valores entre 1 e 10 em quantidade correspondente a num_elementos, e armazene em uma lista chamada elementos. Em seguida imprima:

- A lista elementos
- A soma dos valores da lista
- A média dos valores da lista

3) Preencha duas listas (lista1, lista2) com 20 valores inteiros aleatórios entre 0 a 50. Crie uma terceira lista interseccao contendo apenas os valores que se repetem nas duas listas. Ao final imprima:

- Ambas as listas
- A lista intersecção ordenada
- A quantidade de vezes que cada elemento aparece em cada lista

Atenção, a lista de intersecções não pode ter duplicatas.

```
lista1 - [10, 10, 28, 10, 29, 20, 30, 10, 29, 11]
lista2 - [11, 16, 26, 44, 11, 10, 20, 29, 10, 12]
Interseccao - [10, 11, 20]
Contagem
10: (lista1=4, lista2=1)
11: (lista1=1, lista2=2)
20: (lista1=1, lista2=1)
```

4) Crie um programa em Python que receba duas listas de números do usuário, podendo cada lista ter uma quantidade diferente de valores. Em seguida, combine essas duas listas de forma alternada para formar uma terceira lista. Intercale os elementos até o final da primeira lista, adicionando ao final os elementos remanescentes da maior lista.

Exemplo de interação via terminal (entradas em laranja):

```
Digite a quantidade de elementos da lista 1: 4
Digite os 4 elementos da lista 1:
1
2
3
4
Digite a quantidade de elementos da lista 2: 6
Digite os 6 elementos da lista 2:
5
6
7
8
9
10
Lista intercalada: 1 5 2 6 3 7 4 8 9 10
```

Fatiamento de listas

O fatiamento em listas permite acessar subconjuntos de elementos de uma lista. O fatiamento é feito usando a notação de colchetes **[]** e fornece uma maneira concisa de extrair partes específicas de uma lista. A intuição é parecida com a função **range()**, que aprendemos no módulo de laços de repetição, mas a estrutura da sintaxe é diferente.

lista[inicio : fim : pulo]

Fornecemos até 3 informações dentro de colchetes, separadas pelo símbolo : (dois pontos)

início: Opcional. Índice inicial dos elementos incluídos na fatia [padrão: 0]

fim: Opcional. Índice final dos elementos incluídos na fatia [padrão: len(lista)]

pulo: Opcional. Intervalo entre os índices incluídos na fatia [padrão: 1]

Pense no fatiamento como uma operação mais complexa de indexação, onde no lugar de obter como retorno apenas um elemento, obtemos uma nova lista com o subconjunto indexado.

```
>>> nums = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[2:5] # índices 2 a 5 (início: 2, fim: 5, passo: 1)
[20, 30, 40]
>>> nums[::2] # índices pares (início: 0, fim: len(nums), passo: 2)
>>> [0, 20, 40, 60, 80]
>>> nums[::-1] # lista invertida (início: 0, fim: len(nums), passo: -1)
>>> [90, 80, 70, 60, 50, 40, 30, 20, 10, 0]
```

Podemos usar índices negativos em qualquer um dos parâmetros do fatiamento, sabendo que o caminhamento invertido na lista varia dentro do intervalo **-1** a **-n**. É preciso muito cuidado ao utilizar o **pulo negativo** no fatiamento, já que assumimos um caminhamento na lista invertida. Ao utilizar o pulo negativo, assumimos os seguintes valores padrão para os outros parâmetros:

início: [padrão: fim da lista]
fim: [padrão: início da lista]

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[-4:-1]
[60, 70, 80]
>>> nums[-4:]
[60, 70, 80, 90]
>>> nums[4:-1:2]
[50, 70]
>>> nums[2::-1] # início: 2, final: início da lista, pulo: -1
[30, 20, 10]
nums[:4:-1]     # início: fim da lista, final: 4, pulo: -1
[90, 80, 70, 60]
```

Deletando elementos com del

Agora que já sabemos fatiar listas, podemos conhecer outra forma de deletar elementos, através da palavra reservada **del**. Atuando tanto sobre índices quanto sobre fatias, este comando remove o(s) elemento(s) contidos no índice ou fatia apresentada. Sua sintaxe é:

del lista[i]

del lista[i:j:k]

O comando **del** modifica a lista original e não produz retorno. Mas diferente de outras funções que não produzem retorno, porém na prática retornam um objeto **None**, qualquer tentativa de capturar o retorno deste comando produz um **erro de sintaxe**, como apresentado a seguir.

```
>>> a = [0, 2, 3, 4]
>>> del a[1:3]
>>> a
[0, 4]
>>> x = del a[1:2]
  File "<stdin>", line 1
    x = del a[1:2]
        ^^^
SyntaxError: invalid syntax
```

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula3_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo6 do seu repositório da disciplina no GitHub.

1) Escreva um script em Python que solicita do usuário uma quantidade indefinida de números inteiros (com pelo menos 4 valores), os armazena em uma lista e, usando fatiamento de listas, imprima:

·A lista original
·Os 3 primeiros elementos
·Os 2 últimos elementos
·A lista invertida (do fim para o começo)
·Os elementos de índice par (0, 2, 4 ...)
·Os elementos de índice ímpar (1, 3, 5, ...)

2) Dada uma lista de endereços web (URLs) que sempre começam com "www." e sempre terminam com ".com", use o conceito de fatiamento de listas para criar uma lista dominios com o nome principal de todos os domínios, conforme exemplo a seguir.

```
URLs: ["www.google.com", "www.gmail.com", "www.github.com", "www.reddit.com", "www.yahoo.com"]
dominios: ["google", "gmail", "github", "reddit", "yahoo"]
```

3) Crie uma lista com 20 elementos, entre -10 e 10, gerados aleatoriamente. Em seguida encontre o intervalo que possui a maior quantidade de números negativos e delete ele da lista com o operador del. Você deve imprimir a lista antes e depois da deleção.

```
Original: [9, 2, -1, 4, -2, -3, 5, 6, -7, -4, -1, 6, 8, -3, -6]
Editada: [9, 2, -1, 4, -2, -3, 5, 6, 6, 8, -3, -6]
```

Compreensão de listas

Uma das características mais diferenciadas do Python são seus comandos objetivos e de fácil leitura. Se há uma forma de tornar uma estrutura mais concisa, o Python implementa! Uma das expressões dessa objetividade é o recurso de compreensão de listas, onde **com uma única linha conseguimos construir novas listas.**

A criação de novas listas geralmente é realizada com os seguintes passos:

·Inicializamos uma lista vazia
·Escrevemos um laço de repetição relativo ao problema

•Através de funções como **append** ou operações como concat **(+)** adicionamos um novo elemento a cada iteração.

A compreensão de listas tem uma sintaxe concisa, que consiste em um par de colchetes contendo uma expressão e pelo menos uma operação **for**. Sua saída é a nova lista aplicando a cada elemento o resultado da expressão dada.

resultado = [expressão for elemento in iterável]

•iterável: qualquer objeto iterável, ou seja, capaz de retornar um elemento por vez
•elemento: variável que assume um elemento do iterável a cada iteração
•expressão: qualquer expressão válida que possua retorno. Será executada para cada iteração e geralmente tem relação com o elemento.

A seguir demonstramos a criação (ou transformação) de algumas listas usando a estrutura de laço tradicional e seu equivalente com o recurso de compreensão de listas.

Criando listas com laços	Compreensão de lista equivalente

`sequencia = list()`
`for i in range(10):`
`  sequencia.append(i)`
	`sequencia = [i for i in range(10)]`

`lista = [2, 3, 5, 7]`
`quadrado = []`
`for n in lista:`
`  quadrado.append(n**2)`
	`lista = [2, 3, 5, 7]`
	`quadrado = [n**2 for n in lista]`

`precos = [1.50, 1.99, 19.99]`
`com_juros = []`
`for p in precos:`
`  com_juros.append(p*1.1)`
	`precos = [1.50, 1.99, 19.99]`
	`com_juros = [p*1.1 for p in precos]`

Qualquer expressão única que se visa aplicar aos elementos de uma lista, compondo uma nova lista, pode ser feita com o método compacto. Exemplo: pegar os primeiros elementos em uma lista de listas.

```
>>> lista = [[1,2,3], [4,5,6], [7,8,9]]
>>> items = [ l[0] for l in lista ]
>>> print(items)
[1,4,7]
```

A compreensão de listas também aceita **expressões condicionais**. Essas podem ser incorporadas de duas maneiras:

Filtrando valores indesejados colocando um **if ao final** da compreensão.

```
# retorno = [ expressão for item in lista if condição ]
>>> nums = [-21, 12, 14, -37, -9, 20]
>>> positivos = [ n for n in nums if n > 0 ]
>>> print(positivos)
[12, 14, 20]
```

Aplicando expressões distintas a elementos que atendam determinados critérios, colocando o par **if-else no início** da compreensão.

```
# retorno = [exp_true if condição else exp_false for item in lista ]
>>> nums = [-21, 12, 14, -37, -9, 20]
>>> filt = [ l if l >= 0 else 0 for l in nums ]
>>> print(filt)
[0, 12, 14, 0, 0, 20]
```

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula4_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo6 do seu repositório da disciplina no GitHub.

Escreva um script python que use compreensão de listas para criar as seguintes listas:

•Todos os números pares entre 20 e 50, ou seja, [20, 22, 24, ..., 48, 50]
•Os quadrados de todos os valores da lista [1,2,3,4,5,6,7,8,9]
•Todos os números entre 1 e 100 que sejam divisíveis por 7
•Para todos os valores em range(0,30,3), escreva "par" ou "ímpar" dependendo da paridade do elemento (ex: ['par', 'impar',... , 'impar'])

2) Solicite uma frase do usuário e usando compreensão de listas imprima:

•A lista de vogais da frase, ordenada alfabeticamente
•A lista de consoantes da frase (remova espaços em branco)

```
Digite uma frase: Eu gosto de programar em Python
Vogais: ['a', 'a', 'e', 'e', 'o', 'o', 'o', 'o', 'u']
Consoantes: ['E', 'g', 's', 't', 'd', 'p', 'r', 'g', 'r', 'm', 'r', 'm', 'P', 'y', 't', 'h', 'n']
```

3) Reescreva o código a seguir para construir a lista pagamentos usando compreensão de listas, ou seja, eliminando o laço de repetição.

```
horas_trabalhadas = [40, 37, 45, 40, 40, 48]
ganho_por_hora = 20
hora_extra = 25
pagamentos = []
for hora in horas_trabalhadas:
    pagamentos.append(ganho_por_hora * min(hora, 40) + \
                      hora_extra * max(0, hora-40))
print(pagamentos)
```

4) Reescreva o código a seguir para construir a lista aprovados usando compreensão de listas, ou seja, eliminando o laço de repetição.

```
alunos = ["Maria", "Jose", "Carla", "Sol"]
notas = [35, 50, 20, 80]
aprovados = []
for i in range(len(notas)):
        if notas[i] >= 60:
                aprovados.append(alunos[i])
```

Map e Filter

Assim como a compreensão de listas é uma forma reduzida de realizar operações com elementos de um iterável, as funções **map e filter** também oferecem poderosos recursos para **aplicar funções a elementos de um ou mais iteráveis.**

Map

Função embutida que aplica uma função a um ou mais iteráveis, seguindo a sintaxe:

map(**função, iterável1, iterável2, ..., iterável_n**)

O número de iteráveis fornecido para o map é definido pelo número de parâmetros que **função** requer. Assuma as duas funções a seguir. A função quadrado recebe um único parâmetro e o eleva ao **quadrado**, já a função **soma** recebe dois parâmetros e os soma. Ao aplicá-las com o **map**, fornecemos um único iterável para aplicar a função **quadrado**, a qual será aplicada a cada elemento do iterável. Já para **soma** fornecemos dois iteráveis, e o map caminhará simultaneamente em ambos, alimentando cada par de elementos para **soma**.

```
>>> def quadrado(x): return x**2
...
>>> lista = [1,2,3]
>>> res = map(quadrado, lista)
>>> for r in res: print(r)
```

```
>>> def soma(x,y): return x+y
...
>>> lista1 = [1,2,3]
>>> lista2 = [4,5,6]
>>> res = map(soma, lista1, lista2)
```

```
...
1
4
9
```

```
>>> for r in res: print(r)
...
5
7
9
```

A seguir apresentamos quatro diferentes maneiras de realizar a mesma tarefa, transformar um iterável (nesse caso uma lista), elevando todos os seus elementos ao quadrado. Note que o map trabalha com qualquer função válida, seja customizada, lambda ou qualquer outra.

```
lista = [2, 3, 5, 7]
quadrado = []
for n in lista:
    quadrado.append(n**2)
```

```
lista = [2, 3, 5, 7]
quadrado = [n**2 for n in lista]
```

```
def quadrado(n): return n**2
lista = [2, 3, 5, 7]
quadrados = map(quadrado, lista)
```

```
lista = [2, 3, 5, 7]
quadrado = map(lambda n:n**2, lista)
```

A título de ilustração, vamos realizar a mesma operação usando a função nativa **pow(x, y)**, que recebe dois parâmetros e retorna a potência xy. Nesse caso temos que atentar para a regra do **map**, que deve receber um número correspondente **de iteráveis**. Como apresentado a seguir, dada uma lista que deve ser elevada ao quadrado, construímos uma segunda lista de mesmo tamanho repetindo o número 2 (a potência desejada).

```
>>> lista = [2, 3, 5, 7]
>>> list(map(pow, lista, [2]*len(lista)))
[4, 9, 25, 49]
```

Vale destacar que caso os iteráveis fornecidos não tenham o mesmo tamanho, **a aplicação do map se dará até o final do menor entre os iteráveis.**

Note que no último código convertemos o resultado do **map** para uma lista antes de apresentá-lo. A sintaxe **list(x)**, assim como **int(x)** ou **float(x)** atua como uma conversão de tipo, mas no caso de **list(x)** o parâmetro esperado é um iterável. Vamos entender agora o retorno do **map**, que também destaca sua principal vantagem.

O retorno do **map** está fortemente atrelado ao conceito de **objeto gerador**, um tipo de iterável do Python. Seu retorno é um **iterador**, um objeto capaz de produzir os elementos da sequência item a item quando solicitado em uma iteração, mas **não armazena todo o conteúdo em memória simulataneamente**. Uma velha conhecida nossa, a função **range**, também retorna um iterador, portanto só acessamos seus elementos individualmente via laço de repetição ou se a convertermos, por exemplo, em lista.

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> for i in range(5): print(i, end = " ")
...
0 1 2 3 4
```

O **map** funciona da mesma forma, se tentarmos visualizar seu retorno original, vemos apenas uma menção ao objeto **map** e seu endereço de memória. **Os elementos individuais ainda não existem em memória, ou seja, ainda não estão ocupando recursos do sistema.** Apenas ao iterar no retorno do map acessamos seus elementos.

```
>>> lista = [2, 3, 5, 7]
>>> retorno = map(pow, lista, [2]*len(lista))
>>> retorno
<map object at 0x102d57850>
>>> for r in retorno: print(r, end=" ")
...
4, 9, 25, 49
```

Esse recurso é muito interessante, por exemplo, se trabalharmos com estruturas de dados massivas (ou virtualmente infinitas), cujo carregamento simultâneo em memória seria inviável. Em casos específicos também há ganho computacional, já que não há o gargalo de produzir todos os elementos na chamada do **map**.

Objetos Geradores

Vale a leitura desse tutorial do Real Python para conhecer mais sobre funções e expressões geradoras. A seguir temos uma versão (super) resumida da explicação.

No código ao lado temos a função geradora sequencia_infinita() que inicializa uma variável num e a cada iteração "retorna" o estado da variável e a incrementa. O retorno usa a palavra reservada yield, onde estaria o return, indicando que **a função não se encerra naquele ponto**, apenas envia um valor para quem a chamou, guarda seu estado e espera a próxima iteração. Ela é portanto infinita, e caso você execute o código ao lado, preste atenção! **Você deve interromper a execução manualmente.**

```
# função geradora
>>> def sequencia_infinita():
...     num = 0
...     while True:
...         yield num
...         num += 1
...
...
>>> sequencia_infinita()
<generator object <genexpr> at 0x102c2b6b0>
>>> for i in sequencia_infinita():
...     print(i, end=" ")
...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
[...]
6157834 6157835 6157836 6157837
6157838 6157839 6157840 6157841
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Já as expressões geradoras se parecem muito com a sintaxe da compreensão de listas, mas usa parênteses em vez de colchetes. Assim como funções geradoras, elas retornam objetos iteráveis que apenas produzem elementos quando em uma iteração ou se convertidos em outros tipos, como listas.

```
KeyboardInterrupt

# expressão geradora
>>> nums = (n**2 for n in
...         sequencia_infinita())
>>> nums
<generator object <genexpr> at
0x107fbbc78>
```

Filter

Fica muito mais fácil entender o **filter** uma vez que conhecemos o **map**, já que o papel do **filter** também é aplicar uma função a um iterável e retornar um objeto gerador, além de possuir uma sintaxe muito similar:

filter(função, iterável)

Observando a sintaxe já vemos a primeira diferença, o **filter** aplica a função a um único iterável. Outra importante diferença é a natureza da função esperada, que deve retornar um valor booleano. Isso porque o **filter,** como o nome sugere, **vai filtrar elementos do iterável que satisfazem uma determinada condição estabelecida pela função dada.**

O **filter** usa a função dada para testar a validade de um elemento e retornar esse mesmo elemento caso ele passe no teste. A seguir vemos a solução para o problema de selecionar apenas os números positivos de uma lista implementado de quatro maneiras: laço tradicional, compreensão de listas, **filter** com função customizada e **filter** com função lambda.

```
lista = [-2, 3, -5, 7]
positivos = []
for n in lista:
    if n > 0:
        positivos.append(n)
print(positivos)
```

```
lista = [-2, 3, -5, 7]
positivos = [n for n in lista if n > 0]
print(positivos)
```

```
def is_positivo(n): return n > 0

lista = [-2, 3, -5, 7]
pos = filter(is_positivo, lista)
print(list(pos))
```

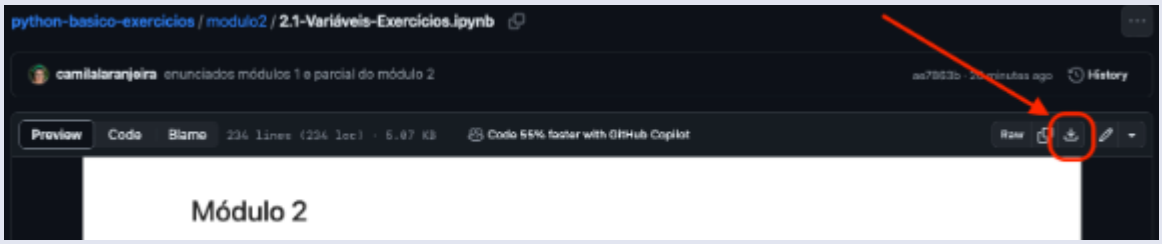
```
lista = [-2, 3, -5, 7]
quadrado = filter(lambda n: n>0,
lista)
print(list(positivos))
```

Note que o retorno booleano da condição não é o retorno final da operação, apenas utilizado para testar os elementos, retornando apenas aqueles cuja condição foi verdadeira.

O restante das propriedades do map se aplicam também ao filter. Ambos retornam iteradores por eficiência de uso de recursos computacionais, e temos portanto que iterar sobre seu resultado ou convertê-lo em outro objeto iterador como a lista. Mas enquanto o map aplica transformações que envolvam um ou mais iteráveis, o filter aplica testes de validade aos elementos de um único iterável, retornando os elementos que passem no teste.

Exercício

Baixe o arquivo 6.5-Listas_Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.

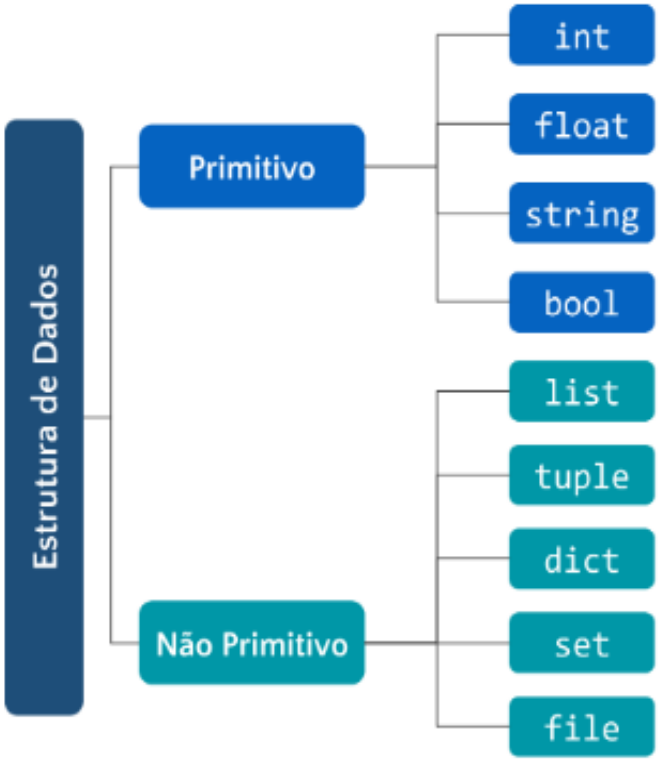


Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo6 do seu repositório.

Strings e Arquivos

Introdução a strings

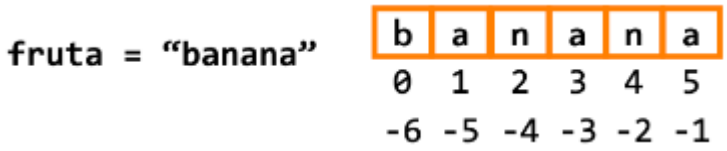
Apesar de serem consideradas um tipo primitivo, assim como **int**, **float** e **bool**, as **strings** são **coleções** de caracteres textuais (letras, números, símbolos ou espaçamentos). Representar texto é uma capacidade fundamental de linguagens de programação. No Python o objeto **string** cumpre esse papel, oferecendo uma variedade de funcionalidades que facilitam a manipulação e formatação de texto. Nesta e nas próximas aulas vamos aprender não só a manipular esse tipo de dado, como a sua aplicação ao lidar com arquivos de texto no computador.



Já conhecemos as strings desde o início do curso por conta de funções como **input()** e **print()**. A representação de uma string no código utiliza aspas simples (') ou duplas ("), assim o interpretador Python está ciente que não se trata de nomes definidos como variáveis ou funções, mas sim textos que serão manipulados ou apresentados em tela.

```
>>> print('Eu sou uma string')
'Eu sou uma string'
>>> print("Eu tambem")
'Eu tambem'
>>> print(Eu nao)
  File "<stdin>", line 1
    print(Eu nao)
          ^^^^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

Por ser uma coleção, uma parte do que aprendemos sobre listas vai se aplicar aqui também (mas nem tudo!). Mas diferente de listas, que são heterogêneas, a unidade atômica de uma string é sempre um caractere textual. Ao inicializar uma string, podemos separar seus elementos em índices da seguinte maneira.



Assim como a lista, a string também é **ordenada**, ou seja, seus elementos possuem índices que representam a ordem que eles foram adicionados (variando de 0 a n-1 ou de -1 a -n). Porém a string **não é mutável** e nem heterogênea.

•**Imutável:** Não é possível modificar os caracteres já definidos na string (existem funções específicas pra isso), ou seja, não se altera diretamente uma string via indexação.
•**Homogênea:** Uma string só pode conter caracteres textuais

Para indexar uma string, acessando seus caracteres individualmente, utilizamos a mesma sintaxe de colchetes que aprendemos com as listas. Podemos utilizar dentro dos colchetes qualquer expressão que retorne um valor inteiro. Note que o valor precisa estar dentro dos limites de começo e final da string, caso contrário, teremos um IndexError (erro de índice).

```
fruta = "melao"
print(fruta[0])
print(fruta[-2])
print(fruta[-4])
tam = len(fruta)
print(fruta[tam-1])
print(fruta[6])
```

```
m
a
e
o
IndexError: string index out of range
```

Strings também dão suporte à forma mais complexa de indexação, o fatiamento. Seu funcionamento é idêntico ao que aprendemos em listas, considerando neste caso os caracteres como unidade atômica da coleção.

fruta = "melao" print(fruta[:3]) print(fruta[2:]) print(fruta[::-1])	mel lao oalem

Strings também são iteráveis, ou seja, podemos caminhar elemento a elemento (caractere a caractere) através de um laço de repetição. Os trechos de código a seguir apresentam a iteração no conteúdo de uma string, assim como em seus índices. Note que, para iterar nos índices da string, fomos capazes de aplicar a função embutida len(), que retorna a quantidade de elementos (caracteres) do objeto.

fruta = "banana" for letra in fruta: print("Letra: %s" % letra)	Letra: b Letra: a Letra: n Letra: a Letra: n Letra: a
fruta = "banana" for i in range(len(fruta)): print("Índice: %d, Letra: %s" % (i, fruta[i]))	Índice 0, Letra: b Índice 1, Letra: a Índice 2, Letra: n Índice 3, Letra: a Índice 4, Letra: n Índice 5, Letra: a

Por ser **imutável**, no entanto, a indexação é uma operação somente leitura. Receberemos um TypeError ao tentar modificar um elemento já inicializado de uma string. Conheceremos mais à frente neste módulo maneiras alternativas de editar uma string.

```
>>> fruta = "banana"
>>> fruta[2] = 'c'
TypeError: 'str' object does not support item assignment
```

Por outro lado, as strings também aumentam de tamanho dinamicamente, dando suporte à operação de concatenação com o operador soma (+). Assim como nas listas, a concatenação só acontece entre objetos do mesmo tipo, neste caso duas strings. Note no exemplo a seguir que espaços em branco não são adicionados automaticamente (é um erro comum no início). Temos que incluir todos os símbolos e espaços desejados. Aos poucos conheceremos melhor essa simbologia.

```
nome = "Clara"
sobrenome = "Oswald"
nome_completo = nome + sobrenome
print("Ola", nome_completo)
```

```
Ola ClaraOswald
```

Já o operador aritmético da multiplicação (*) opera uma string e um inteiro, concatenando múltiplas cópias da mesma string.

```
>>> print("-" * 10)
'----------'
```

A palavra reservada **in** também se aplica às strings como operador de checagem de elementos. Neste caso podemos buscar não só caracteres únicos como também **substrings**, partes da string. Se a palavra reservada **not** é utilizada (precedendo a palavra in ou no início da expressão), negamos o resultado da checagem, invertendo seu valor booleano.

```
False
>>> not "ba" in nome
True
```

É possível comparar strings com operadores relacionais como > (maior que), < (menor que), que são nativamente numéricos. Nesse caso, considera-se a **ordem alfabética entre as strings.** Por exemplo, na comparação entre as strings Romeu > Julieta, Romeu será considerado maior que Julieta já que R é alfabeticamente maior que J.

```
>>> print("Romeu" > "Julieta")
True
>>> print("L" <= "l")
True
```

Na verdade, se considera a ordem da tabela Unicode incluindo letras, números, pontuação e caracteres de controle (enter, del, tab, etc.). Isso porque internamente, caracteres são mapeados para números dessa tabela. Ou seja, tudo que discutimos nessa aula se aplica a todos os caracteres válidos.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r

19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

As funções **chr()** e **ord()** permitem converter números em caracteres e caracteres em números de acordo com a tabela Unicode. Esses números são usados por operações nativamente numéricas ao trabalhar com strings.

```
>>> ord("L")
76
>>> chr(76)
'L'
>>> chr(76+2)
'N'
```

Já conhecemos funções embutidas que trabalham com iteráveis (como min, max e sorted). Ao operar sobre strings, essas funções consideram a unidade atômica do caractere e os valores numéricos da tabela Unicode para realizar suas operações.

```
>>> max("aeiouAEIOU")
'u'
>>> min("aeiouAEIOU")
'A'
>>> sorted("aeiouAEIOU")  # nem tudo retorna string
['A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u']
```

Já quando temos strings como membros de listas, e utilizamos as mesmas funções embutidas, considera-se o elemento atômico da lista como toda a string. Veja os exemplos a seguir.

```
>>> max(["a", "b", "c"])
c
>>> min(["William", "Romeu", "Julieta"])
Julieta
>>> sorted(["William", "Romeu", "Julieta"])
['Julieta', 'Romeu', 'William']
```

Essa dualidade da string como tipo primitivo e ao mesmo tempo coleção de dados foi o que nos motivou a adiar esse tópico na ementa. Agora que você já conhece os dois conceitos (dados primitivos e não primitivos) é possível explorar todas as nuances do tipo string.

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula1_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo7 do seu repositório da disciplina no GitHub.

1) Escreva um programa que solicita o nome do usuário e o imprime em forma de escada, como indicado no exemplo a seguir.

```
Digite seu nome: Fulano
F
Fu
Ful
Fula
Fulan
Fulano
```

2) Escreva um programa que solicite ao usuário inserir seu primeiro nome e sobrenome separadamente. Em seguida, concatene essas duas strings e exiba a mensagem de boas-vindas.

```
Digite seu primeiro nome: Alice
Digite seu sobrenome: Silva
Bem-vinda, Alice Silva!
```

3) Escreva um script que dado uma frase conta os espaços em branco.

```
Digite a frase: Meu amor mora em Roma e me deu um ramo de flores
Espaços em branco: 11
```

4) Faça um programa que leia um número de celular e, caso o número tenha apenas 8 dígitos, acrescente o 9 na frente. Caso o número já tenha 9 dígitos, verifique se o primeiro dígito é 9. Adicione o separador "-" na sua impressão.

```
Digite o número: 97651234
Número completo: 99765-1234
Digite o número: 980876543
Número completo: 98087-6543
```

5) Implemente um código que leia uma string do usuário e imprima quantas vogais existem na frase e quais os seus índices da string. Dica: letra in "aeiou". Exemplo:

```
Digite uma frase: Meu amor mora em Roma e me deu um ramo de flores
19 vogais
Índices [1, 2, 4, 6, 10, 12, 14, 18, 20, 22, 25, 28, 29, 31, 35, 37, 40, 44, 46]
```

6) Dada uma string e uma palavra objetivo, encontre todos os anagramas da palavra objetivo. Anagramas são palavras com os mesmos caracteres rearranjados.

```
Digite uma frase: Meu amor mora em Roma e me deu um ramo de flores
Digite a palavra objetivo: amor
Anagramas: ["amor", "mora", "ramo", "Roma"]
```

7) Crie a função encrypt() que recebe uma lista de strings e retorna os nomes criptografados, bem como a chave da criptografia. Regras:

•Chave de criptografia: gere um valor n aleatório entre 1 e 10
•Substitua cada caracter c pelo caracter c + n. Trabalharemos apenas com o intervalo de caracteres visíveis (entre 33 e 126 na tabela Unicode)

```
nomes = ["Luana", "Ju", "Davi", "Vivi", "Pri", "Luiz"]
chave_aleatoria = 5
nomes_cript = ['Qzfsf', 'Oz', 'If{n', '[n{n', 'Uwn', 'Qzn!']
```

8) Desenvolva um validador de CPF. Solicite do usuário um CPF na forma XXX.XXX.XXX-XX (lido como string) e imprima "Válido" ou "Inválido".

O primeiro passo é calcular o **primeiro dígito verificador**. Separamos os primeiros 9 dígitos do CPF (ex: 111.444.777) e multiplicamos cada um dos números, da direita para a esquerda por números crescentes a partir do número 2, como no exemplo abaixo:

CPF	1	1	1	4	4	4	7	7	7
Multiplicador	10	9	8	7	6	5	4	3	2
Resultado	10	9	8	28	24	20	28	21	14

Em seguida somamos o resultado: 10+9+8+28+24+20+28+21+14 = 162

Pegamos o resultado e dividimos por 11. 162 / 11 = 14 com resto 8

• Se o resto da divisão for menor que 2, então o dígito é igual a 0 (Zero).
• Se o resto da divisão for maior ou igual a 2, então o dígito verificador é igual a 11 menos o resto da divisão (11 - resto).

No nosso exemplo temos que o resto é 8 então faremos 11-8 = 3. Logo o primeiro dígito verificador é 3. Então sabemos que o CPF deve ser: 111.444.777-3X

Para calcular o segundo dígito vamos usar o primeiro dígito já calculado. Vamos montar a mesma tabela de multiplicação usada no cálculo do primeiro dígito. Só que desta vez usaremos na segunda linha os valores 11,10,9,8,7,6,5,4,3,2 já que estamos incluindo mais um dígito no cálculo:

CPF	1	1	1	4	4	4	7	7	7	3
Multiplicador	10	9	8	7	6	5	4	3	2	2
Resultado	10	9	8	28	24	20	28	21	14	6

Somamos: 11 + 10 + 9 + 32 + 28 + 24 + 35 + 28 + 21 + 6 = 204
Dividimos o total do somatório por 11 e consideramos o resto da divisão.
204 / 11 = 18 e resto 6

Aplicamos a mesma regra que utilizamos para obter o primeiro dígito:
- Se o resto da divisão for menor que 2, então o dígito é igual a 0 (Zero).
- Se o resto da divisão for maior ou igual a 2, então o dígito é igual a 11 menos o resto da divisão (11 - resto).

11-6 = 5, logo 5 é o nosso segundo dígito verificador e o CPF completo é:

111.444.777-35

O CPF de entrada deve ser considerado válido se os dígitos fornecidos pelo usuário forem os mesmos dígitos calculados através do processo acima.

Exemplos válidos	Exemplos inválidos
545.315.761-52	545.315.761-52
473.063.662-70	473.063.662-98
775.682.566-77	775.682.566-13

Operações e funções de strings

Usando a função embutida **dir()** é possível listar todas as funções membro de uma string. Destacado em laranja estão as funções que podem ser utilizadas através da sintaxe **string.funcao().** Vamos conhecer a seguir algumas funções mais populares e seus papéis.

```
>>> nome = 'Vitor Augusto'
>>> dir(nome)
['__add__', '__class__', ..., '__subclasshook__' ,'format', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'isspace', 'islower', 'isupper', 'join', 'lower',
'split', 'upper']
```

frase.index(str)

Retorna a primeira ocorrência da string str na frase.

frase.count(str)

Retorna a quantidade de ocorrências da string str na frase.

frase.replace(str1, str2)

Retorna a frase trocando todas as ocorrências da str1 pela str2.

```
>>> frase = "O rato roeu a roupa"
>>> frase.index("o")
5
>>> frase.count("ro")
2
>>> frase.replace("ro", "**")
'O rato **eu a **upa'
```

Existem duas funções muito úteis, **join** e **split**, que atuam com o conceito de **separador**, uma string de separação que pode ser usada para unir uma lista de strings (join = unir) ou como referência para separar uma string (split = separar), transformando-a em uma lista.

separador.join(list)

Recebe uma lista de strings e retorna uma única string unindo os elementos com o separador. Note que a chamada é realizada pela string separadora.

string.split(separador)

Reparte a string que chamou a função, de acordo com o separador fornecido, e retorna uma lista. Se nenhum separador é fornecido, considera-se o espaço em branco.

```
>>> frase = "O rato roeu a roupa"
>>> palavras = frase.split()
>>> palavras
["O", "rato", "roeu", "a", "roupa"]
>>> "__".join(palavras)
'O__rato__roeu__a__roupa'
>>> "".join(['C', 'r', 'i', 's'])
'Cris'
>>> nomes = "Cris Souza, Alice Sá"
>>> nomes.split(',')
['Cris Souza', ' Alice Sá']
```

Tem duas coisas interessantes nos exemplos dados acima. Primeiro, o uso da string vazia ("""), duas aspas sem nenhum caractere interno, que pode ser utilizada para unir uma lista de strings sem nenhum separador. Além do último comando split, onde utilizamos o caractere "," para separação e tivemos como consequência um espaço em branco no início do segundo nome. Em um problema real, teríamos que tratar detalhes como esse para formatar as strings de maneira apropriada.

Existem muitas outras funções membro de string que não vale a pena repetir nessa apostila, já que estão muito bem descritas na documentação. A exemplo de funções de:

Formatação	Classificação
frase.lower() Retorna todos os caracteres da frase em minúsculo	**frase.islower()** Retorna True se todos os caracteres da string são minúsculos, False caso contrário.

frase.upper()

Retorna todos os caracteres da frase em maiúsculo

frase.capitalize()

Retorna o primeiro caracter em maiúsculo e os outros minúsculos

frase.isalpha()

Retorna True se todos os caracteres da string são alfabéticos, False caso contrário.

frase.isdigit()

Retorna True se todos os caracteres da string são dígitos numéricos, False caso contrário.

Por fim, mas não menos importante, podemos aplicar as funções **map** e **filter** em strings. Como vimos anteriormente, elas produzem objetos geradores como resultado. Esses infelizmente não podem ser diretamente convertidos para strings se quisermos acessar todos os elementos simultaneamente. Uma possível solução seria converter para lista e em seguida usar o join para unir em uma única string. Mas em geral usaremos o map em casos onde é vantajoso trabalhar com o objeto gerador produzido pelo map.

```
def encrypt(c, key, minchr=33, maxchr=127):
  nc = ord(c)+key
  return chr(nc) if nc < maxchr else
        chr(minchr + (nc % maxchr))

nome = "Priscila"
codif = map(encrypt, nome, [2]*len(nome))
print("".join(list(codif)))
```

```
'Rtkueknc'
```

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula2_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo7 do seu repositório da disciplina no GitHub.

1) Faça um programa que solicite a data de nascimento (dd/mm/aaaa) do usuário e imprima a data com o nome do mês por extenso. Dica: usando listas você não precisa fazer um "if" para cada mês.

```
Digite uma data de nascimento: 29/10/1973
Você nasceu em  29 de Outubro de 1973.
```

2) Desenvolva um programa que solicite ao usuário inserir uma frase e substitua todas as ocorrências de vogal por "*".

```
Digite uma frase: O rato roeu a roupa do rei
Frase modificada: * r*t* r*** * r**p* d* r**
```

3)Desenvolva um programa que verifique se uma frase fornecida pelo usuário é um palíndromo (ou seja, lida da mesma forma de trás para frente). Ignore espaços em branco ou sinais de pontuação, e considere maiúsculas e minúsculas da mesma forma. **Seu programa deve continuar rodando até que o usuário digite "Fim".**

```
Digite uma frase (digite "fim" para encerrar): Radar
"Radar" é palíndromo
Digite uma frase (digite "fim" para encerrar): Bom dia!
"Bom dia!" não é palíndromo
Digite uma frase (digite "fim" para encerrar): Ame o poema
"Ame o poema" é palíndromo
Digite uma frase (digite "fim" para encerrar): A Daniela ama a lei? Nada!
"A Daniela ama a lei? Nada!" é palíndromo
Digite uma frase (digite "fim" para encerrar): fim
```

4) Implemente uma função em Python chamada validador_senha() que verifica se uma senha fornecida atende **todos** os seguintes critérios:

- Pelo menos 8 caracteres de comprimento.
- Contém pelo menos uma letra maiúscula e uma letra minúscula.
- Contém pelo menos um número.
- Contém pelo menos um caractere especial (por exemplo, @, #, $).

```
def validador_senha(senha):
    #### Escreva a função

# Exemplo de uso:
senha1 = "Senha123@"
senha2 = "senhafraca"
senha3 = "Senha_fraca"
print(validador_senha(senha1))  # Saída esperada: True
print(validador_senha(senha2))  # Saída esperada: False
print(validador_senha(senha3))  # Saída esperada: False
```

5) Implemente uma função chamada embaralhar_palavras() que recebe uma frase como entrada e retorna uma nova frase com as letras internas de cada palavra embaralhadas. **Mantenha sempre** o primeiro e último caractere da palavra no lugar.

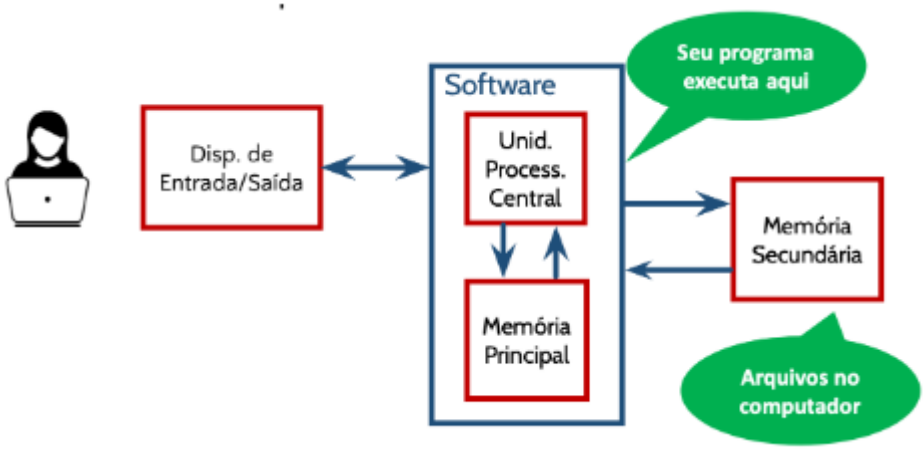
Dica: use a biblioteca random.

```
def embaralhar_palavras(frase):
    #### Escreva a função

# Exemplo de uso:
frase = "Python é uma linguagem de programação"
resultado = embaralhar_palavras(frase)
print(resultado)
# Possível saída: "Ptohyn é uma lignaugem de prarmoagãço"
```

Acessando arquivos

Até agora, trabalhamos com entrada de dados pelo console, usando a função **input()**. Esses dados existem de maneira provisória, apenas durante a execução do seu script. São armazenados na memória principal do computador (RAM) para rápido acesso durante o processamento. Por outro lado, arquivos armazenados na memória secundária (HD) são salvos de forma permanente, persistindo mesmo que você encerre o programa ou desligue o computador. Aprendendo a manipular esses arquivos com Python, podemos trabalhar com dados mais extensos e complexos, como textos, tabelas, imagens, páginas web e outros.



Nessa aula conheceremos 4 importantes elementos para manipulação de arquivos:

Módulo os: comandos do sistema operacional (operating system)

Módulo sys: interações com o ambiente de execução

Input/Output: manipuladores de arquivo nativos do python

Instrução with: instrução composta para simplificar a manipulação de arquivos

Para facilitar a compreensão dessa aula, é importante ter assistido as aulas de manipulação de arquivos e manipulação de diretórios do **curso de Linux.**

Módulo os

O módulo os, que pode ser importado em scripts Python, cria uma interface de comunicação com o sistema operacional, permitindo interações como navegar nas pastas do sistema, manipular diretórios e arquivos, entre outros. Falaremos especificamente destas capacidades, caso você deseje saber mais, consulte a documentação.

Quando executamos um script python, essa execução acontece em um local específico da árvore de diretórios. Você pode usar o comando **pwd** no terminal, antes de executar um script, para consultar o local em seu computador onde a execução vai acontecer. Pode também usar o comando ls para listar todos os arquivos presentes na pasta atual. Veja a seguir.

```
mila@ubuntu:~/Github/python-basico/modulo7$ pwd
/Users/milalaranjeira/Github/python-basico/modulo7
mila@ubuntu:~/Github/python-basico/modulo7$ ls
aula1_questao1.py    aula1_questao2.py
aula1_questao3.py    aula1_questao4.py
aula1_questao5.py    script.py
mila@ubuntu:~/Github/python-basico/modulo7$ python3 script.py
```

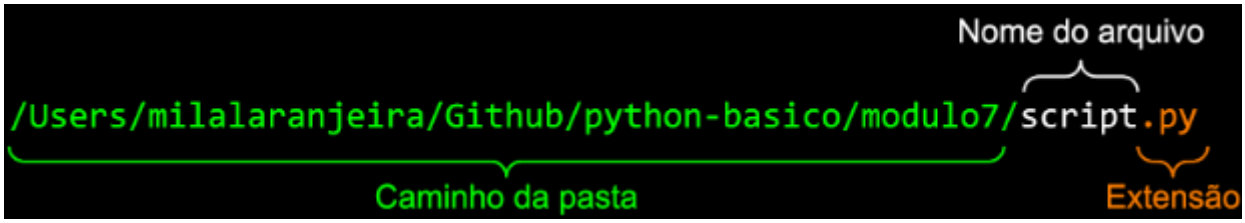
Os arquivos que desejamos manipular dentro de nosso script Python pode não estar localizado na mesma pasta onde o código foi executado, por isso precisamos do módulo **os** para interagir e manipular diretórios e arquivos do sistema.

No código ao lado, importamos **os** e demonstramos a funcionalidade de alguns dos métodos mais comuns:

- **os.getcwd():** retorna uma string com o caminho da pasta atual.
- **os.listdir(path):** lista as pastas e arquivos localizados em **path**.
- **os.mkdir(path):** cria um diretório chamado **path**.
- **os.rename(src, dst):** renomeia o arquivo ou diretório **src** para **dst**.
- **os.rmdir(path):** exclui o diretório **path.**

```
mila@ubuntu:~$ python3
>>> import os
>>> os.getcwd()
'/home/mila/'
>>> type(os.getcwd())
<class 'str'>
>>> os.curdir
'.'
>>> os.mkdir('nova_pasta')
>>> 'nova_pasta' in os.listdir(os.curdir)
True
>>> os.rename('nova_pasta', 'pasta')
>>> 'pasta' in os.listdir(os.curdir)
False
>>> os.rmdir('pasta')
```

Note que os caminhos do sistema são sempre strings, seguindo a padronização do sistema operacional. No Linux, pastas são divididas por uma barra '/', e a estrutura do caminho de um arquivo possui 3 elementos:



•Caminho da pasta: o caminho até a pasta onde o arquivo está localizado. Nomes de pasta são sempre seguidos por uma barra "/" (Linux e Mac) ou barra invertida "\" (Windows).

•Nome do arquivo: o nome do arquivo.

•Extensão: o final do caminho do arquivo pré-anexado com um ponto (.) usado para indicar o tipo de arquivo.

os.path

O módulo **os** possui um submódulo **path** que implementa funcionalidades especializadas em nomes de caminhos do sistema. Vá na documentação do módulo os.path e consulte a descrição das funções usadas a seguir. É bom criar o hábito desde já :)

```
mila@ubuntu:~$ python3
>>> import os
>>> os.listdir('.')
['Pictures', 'Desktop', 'Documents', 'Downloads', 'arquivo.txt']
>>> name = os.path.abspath('arquivo.txt')
>>> name
'/home/mila/arquivo.txt'
>>> os.path.split(name)
('/home/mila/', 'arquivo.txt')
```

```
(os.path.basename(name))
'arquivo.txt'
>>> os.path.splitext
('arquivo', 'txt')
>>> os.path.isfile('arquivo.txt')
True
>>> os.path.isfile('outro_arquivo.txt')
False
>>> os.path.isfile('Downloads')
False
>>> os.path.isdir('Downloads')
True
>>> os.path.exists('Downloads')
True
>>> os.path.exists('outro_arquivo.txt')
False
```

Módulo sys

Permite manipular o ambiente de execução, interagindo com o interpretador. No exemplo a seguir consultamos detalhes da versão do interpretador Python ativo, como o número e a data de compilação.

```
mila@ubuntu:~$ python3
>>> import sys
>>> sys.version # versão do Python
'3.11.6 (main, Oct  2 2023, 13:45:54) [Clang 15.0.0 (clang-1500.0.40.1)]
```

Para nós nesse momento, a principal funcionalidade do módulo **sys** é receber argumentos de execução de um script. Quando executamos um script via linha de comando, o nome do script é um argumento, um parâmetro, passado para que o interpretador saiba qual script executar. Podemos passar argumentos adicionais e recebê-los dentro do script através da variável **argv** do módulo **sys**. Essa variável é um objeto do tipo lista, onde cada elemento é uma string referente aos argumentos digitados, sendo a primeira string **(sys.argv[0])** o nome do script.

No código a seguir fazemos duas checagens importantes:

•Se existem argumentos a partir do índice 1 da lista em **argv.**

•Se o nome fornecido é um arquivo que existe no sistema

Essas checagens sempre devem ser feitas para garantir que trabalharemos com uma informação válida de arquivo. Caso contrário, chamamos **sys.exit()** encerrando a execução.

```
mila@ubuntu:~$ python3 script.py arq1.txt
```

```
# script.py
import os, sys

argumentos = sys.argv[1:]
if not argumentos:
        print('Informe o nome do arquivo!')
        sys.exit()

arquivo = argumentos[0]
if not os.path.isfile(arquivo):
        print(f'Arquivo {arquivo} não existe!')
        sys.exit()
```

Assim como no input, todos os argumentos serão lidos como string. Lembre-se de converter! Em módulos futuros conheceremos formas melhores de lidar com isso **(argparse)**.

```
mila@ubuntu:~$ python3 script.py teste 2 3.14
teste <class 'str'>
2 <class 'str'>
3.14 <class 'str'>
```

```
# script.py
import sys

argumentos = sys.argv[1:]
for arg in argumentos:
   print(arg, type(arg))
```

Input/Output

Objetos do tipo **io** (input/output, ou entrada/saída) permitem manipular arquivos. Esse tipo de objeto é retornado pela função **open**, responsável por abrir um arquivo, e seu retorno é nossa referência para todas as operações envolvendo o arquivo aberto. Os principais parâmetros da função **open** são 2 posicionais e 1 nomeado:

```
open(filename, mode, encoding='utf-8')
```

•**filename:** string com o caminho para o arquivo que desejamos abrir
•**mode:** string com o modo de abertura, que pode conter combinações dos modos apresentados na tabela a seguir.

r	Somente leitura (padrão, arquivo precisa existir)
w	Somente escrita (sobrescreve o conteúdo do arquivo, se houver)
a	Somente escrita (anexa ao final do conteúdo do arquivo)
r+	Leitura e escrita
t	Modo texto (padrão)
b	Modo binário

•**encoding:** codificação do conteúdo. Para arquivos de texto utilizaremos normalmente a codificação padrão do Python, de acordo com a tabela unicode (**utf-8**).

No código a seguir, abrimos um arquivo de texto para escrita com intenção de criar um novo arquivo. Como passamos apenas o nome e a extensão, sem um caminho de pasta, ele é criado no local onde a execução acontece. Em seguida, usamos a variável **io** retornada por **open** para escrever no arquivo.

•**f.write(str):** Escreve a string **str** no arquivo **f**. O conteúdo escrito só será salvo em disco rígido **após o arquivo ser fechado**.
•**f.close():** fecha o arquivo **f**.

```
>>> os.listdir(os.curdir)
['Pictures', 'Desktop', 'Documents', 'Downloads']
>>> f = open('arquivo.txt', 'w', encoding='utf-8')
>>> type(f)
<class '_io.TextIOWrapper'>
>>> f.write('Aprendendo a acessar arquivos!')
>>> f.close()
>>> os.listdir(os.curdir)
['Pictures', 'Desktop', 'Documents', 'Downloads', 'arquivo.txt']
```

Com o arquivo criado e seu conteúdo salvo, após o fechamento, podemos abrí-lo para leitura e usar a função read para carregar todo o seu conteúdo como string.

•**str = f.read():** Lê o conteúdo do arquivo **f** e o retorna como string.

```
>>> f = open('arquivo.txt', 'r')
>>> lines = f.read()
>>> print(lines)
'Aprendendo a acessar arquivos!'
>>> f.close()
```

Instrução with

A estrutura comum de manipulação de arquivos sempre começa no **open** e termina no **close**. Para gerenciar esses processos de maneira mais segura e eficiente podemos utilizar a instrução **with**, uma instrução composta que, assim como **if**, **for** ou **while**, controla o fluxo de execução de um bloco interno de código.

```
with open('arquivo.txt', 'r+') as f:
    # bloco interno de comandos
    # de manipulação de arquivos

# programa principal
print(f.closed) # True
```

A instrução **with** se encarrega automaticamente de fechar o arquivo ao final de seu escopo, quando o fluxo de execução sai do seu bloco, mesmo em casos de erro. Você pode consultar o status de **f.closed** fora do bloco **with** para confirmar que o arquivo **f** está fechado.

Arquivos de texto

Um arquivo de texto é uma string que pode ser divida em uma sequência de linhas. Embora não seja visível, no final de cada linha existe um caractere chamado "newline" que indica ao computador o fim de linha, ele é representado como "\n" e lido como um único caractere.

arquivo.txt

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008\n
Return-Path: <postmaster@collab.sakaiproject.org>\n
Date: Sat, 5 Jan 2008 09:12:18 -0500\n
To: source@collab.sakaiproject.org\n
From: stephen.marquard@uct.ac.za\n
Subject: [sakai] svn commit: r39772 - content/branches/\n
\n
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772\n
```

Um identificador de arquivo aberto para leitura é um iterável nas linhas do arquivo. A cada iteração, retorna a string correspondente à próxima linha do arquivo até alcançar o final do arquivo (end of file - **EOF**).

```
f = open('mbox.txt')
for linha in f:
    print(linha)
```

Vimos na aula anterior que a função **read** pode ser utilizada para ler todo o conteúdo de um arquivo como uma única string. Podemos utilizar também a função **readlines**, que já fornece uma lista de strings onde cada elemento é uma linha do arquivo.

```
>>> f = open('mbox.txt')
>>> texto = f.read()
>>> print('No de caracteres:', len(texto))
No de caracteres: 6687002
>>> print(texto[:2])
'Fr'
>>> f = open('mbox.txt')
>>> linhas = f.readlines()
>>> print('No de linhas:', len(linhas))
No de linhas: 132045
>>> print(linhas[:2])
['From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008\n','Return-Path:
<postmaster@collab.sakaiproject.org>\n']
```

Podemos explorar a vasta biblioteca de funções de string para processar o texto de um arquivo. Por exemplo, para filtrar o arquivo, imprimindo apenas as linhas de destinatários (note que é um arquivo de e-mails), podemos usar a função **str1.startswith(str2)**. Essa função checa se a string **str1** inicia com a substring **str2**. Também utilizamos a função **str.rstrip()** que remove todo o espaçamento do lado direito da string **str**, com o objetivo de remover o caractere "\n" já que este é adicionado à impressão pelo **print**.

```
f = open('mbox.txt')
total = 0
for linha in f:
        if linha.startswith('From:'):
    print(linha.rstrip())
    total += 1
f.close()
print(f'Existem {total} e-mails.')
```

```
From: stephen.marquard@uct.ac.za\n
From: louis@media.berkeley.edu\n
From: zqian@umich.edu\n
...
Existem 1797 e-mails.
```

Na hora de escrever em arquivos, temos que tomar alguns cuidados. Não podemos assumir que nenhum espaçamento, incluindo quebra de linha, vai ser adicionado automaticamente. No exemplo a seguir escrevemos acidentalmente um arquivo de uma única linha por esquecer de colocar o caracter "**\n**" ao final de cada string. E vale relembrar que a escrita só é efetivada (armazenada em disco) após o fechamento do arquivo.

script.py	arquivo.txt
f = open('arquivo.txt') f.write('Ola mundo!') f.write('Esse é meu arquivo') f.write('de 3 linhas.') f.close()	Ola mundo!Esse é meu arquivode 3 linhas.

Assim como a função **readlines** retorna uma lista das linhas, a função **writelines** recebe uma lista de strings e as escreve no arquivo. Aqui permanece a regra que adicionar o newline é nossa responsabilidade. No exemplo a seguir adicionamos o "**\n**" ao final de cada elemento usando compreensão de listas.

script.py	arquivo.txt
items = ['laranja', 'manga', 'uva'] items = [i+'\n' for i in items] with open('arquivo.txt', 'w') as f: f.writelines(items)	laranja manga uva

Essa é apenas uma pequena introdução a arquivos de texto. O Python tem uma capacidade muito maior, que pode ser até suplementada através de bibliotecas muito poderosas. Recomendo você consultar as documentações:

- de strings: https://docs.python.org/pt-br/3/library/string.html
- de arquivos: https://docs.python.org/pt-br/3/tutorial/inputoutput.htmlan

Exercício

Instruções para submissão: Para cada questão, crie no VSCode um arquivo chamado aula4_questaoX.py, sendo X o número da questão. Faça commit de todos os arquivos para a pasta modulo7 do seu repositório da disciplina no GitHub.

1) Escreva um script Python que solicita uma frase do usuário e a salve em um arquivo chamado "frase.txt" no mesmo local do seu script. Imprima em seguida o caminho completo do arquivo salvo.

```
Digite uma frase: Bom dia, meu nome é Davi.
Frase salva em /Users/laranjeira/python-basico/frase.txt
```

2) Escreva um script que leia o arquivo salvo no exercício anterior e salva em um novo arquivo "palavras.txt", removendo todos os espaços em branco e caracteres não alfabéticos, e separando cada palavra em uma linha. Ao final, imprima o conteúdo do arquivo "palavras.txt".

```
Bom
dia
meu
nome
é
Davi
```

3) Baixe o arquivo contendo o roteiro do filme brasileiro "Estômago" e salve em seu computador com o nome "estomago.txt". Em seguida crie um script em Python que abra o arquivo para leitura e imprima:

- O texto das primeiras 25 linhas
- O número de linhas do arquivo
- A linha com maior número de caracteres
- O número de menções aos nomes dos personagens "Nonato" e "Íria" (inclua todas as variações de maiúsculas e minúsculas e atenção para não incluir a substring "iria" se ela fizer parte de outras palavras).

4) Vamos fazer o jogo da forca! Antes de programar:

•Crie um arquivo no seu computador chamado "gabarito_forca.txt" com uma lista de 10 palavras de sua escolha (separadas por quebras de linha, "\n"). Essas serão as opções de palavra do jogo.
•Crie um arquivo chamado "gabarito_enforcado.txt" com o conteúdo apresentado ao final dessa questão.

5) Escreva um programa em Python para executar o jogo, de acordo com as definições:
•Abra o arquivo "gabarito_forca.txt" e escolha aleatoriamente uma palavra;
•Com o arquivo "gabarito_enforcado.txt", crie uma lista de strings com os estágios do enforcado;
•No início exiba o número de letras na palavra como underscores;
•Permita que o jogador insira letras para adivinhar a palavra;
•Em caso de acerto, mostre o progresso do jogador substituindo os underscores correspondentes à letra digitada;
•Em caso de erro, crie a função "imprime_enforcado()" que recebe um inteiro indicando o número de erros do jogador e imprime o enforcado correspondente;
•Limite o número de tentativas para 6 (as partes do enforcado).

gabarito_enforcado.txt

```
 |---|
     |
     |
     |
=========

 |---|
 O   |
     |
     |
=========
```

```
 |---|
 O  |
 |  |
    |
=========

 |---|
 O  |
/|  |
    |
=========

 |---|
 O  |
/|\ |
    |
=========

 |---|
 O  |
/|\ |
/   |
=========

 |---|
 O  |
/|\ |
/ \ |
=========
```

5) A extensão ".csv" significa "comma-separated values" ou "valores separados por vírgula". É a extensão utilizada por sistemas de gerência de tabelas como o Microsoft Excel ou Google Sheets. Nesse exercício vamos criar uma planilha com dados sobre livros que você já leu ou gostaria de ler. Siga as instruções.

•Selecione pelo menos 10 livros que você leu ou gostaria de ler. Você

deve reunir as seguintes informações: título, autor, ano de publicação e número de páginas.
•No Python, crie um arquivo chamado "meus_livros.csv", aberto para escrita.
•Na primeira linha escreva os títulos da planilha separados por vírgula (sem espaço em branco). Os títulos são: "Título", "Autor", "Ano de publicação" e "Número de páginas". Lembre de finalizar a linha com uma quebra de linha.
•A partir da segunda linha escreva as informações de cada livro que você levantou, separando cada informação por uma vírgula (sem espaço em branco). Lembre de finalizar cada linha com uma quebra de linha.
•Feche o arquivo para salvá-lo e abra com a ferramenta de planilhas de sua escolha. Como você já tem conta no Google, sugiro abrir com o Google Sheets.

Seu arquivo deve ser aberto como uma planilha parecida com essa:

Título	Autor	Ano de publicação	Número de páginas
O Caçador de Pipas	Khaled Hosseini	2003	368
Torto Arado	Itamar Vieira Junior	2019	264

6) Vamos descobrir as músicas mais populares do Spotify nos últimos 10 anos!
•Crie uma conta no Kaggle, uma das principais plataformas de ciência de dados e aprendizado de máquina. Em disciplinas avançadas vamos trabalhar com bases de dados provenientes de lá!
•Baixe o arquivo spotify-2023.csv no final da página que apresenta os dados.
•No Python, abra o arquivo para leitura e imprima as cinco primeiras linhas
Para abrir o arquivo, defina o parâmetro **encoding='latin-1'**
•Após compreender a estrutura do arquivo (divisão em colunas, caracter separador de coluna, etc.) passamos para a etapa de extração de informações.

O arquivo está estruturado da seguinte forma: cada linha representa uma música e contém as seguintes informações separadas por vírgula (CSV):

track_name,artist(s)_name,artist_count,released_year,released_month,released_day,in_spotify_playlists,in_spotify_charts,streams,in_apple_playlists

Usaremos apenas informações das colunas:

track_name	Nome da música
artist(s)_name	Nome do artista
artist_count	Número de artistas listados em artist(s)_name
released_year	Ano de lançamento
streams	Número de vezes que a música foi tocada no Spotify

Você deve criar um script Python para processar esse arquivo e gerar uma lista com 10 elementos, cada qual representando a música mais tocada de cada ano no intervalo de 2012 a 2022. Considere somente músicas dentro do intervalo solicitado. Cada elemento da lista produzida deve conter as seguintes informações:

[track_name, artist(s)_name, released_year, streams]

Essa atividade tem alguns desafios. Assim como as colunas da tabela são separadas por vírgulas, músicas com mais de um artista (artist_count>1) terá o campo artist(s)_name entre aspas com o nome dos artistas separado por vírgulas. Ex:
Seven (feat. Latto) (Explicit Ver.),"**Latto, Jung Kook**",2,2023, ...

Há também nomes de músicas entre aspas por conter caracteres especiais como vírgulas ou aspas. Ex:

"**Peso Pluma: Bzrp Music Sessions, Vol.55**","Bizarrap,Peso Pluma",2,2023,
Você deve ignorar essas linhas, e terá portanto que propor uma

verificação para identificá-las.
Ao final imprima a lista produzida. Ex:
[['When I Was Your Man', 'Bruno Mars', 2012, 1661187319],
['I Wanna Be Yours', 'Arctic Monkeys', 2013, 1297026226],
...,
['As It Was', 'Harry Styles', 2022, 2513188493]]

Arquivos Binários

Todos os arquivos que não são textuais, são considerados binários. Mas é importante ressaltar que na prática todo arquivo é binário, inclusive textos. A extensão de um arquivo indica como seus bits serão interpretados. Ao abrir um arquivo no modo texto, estamos apenas indicando como interpretar as regras daquela codificação.

Aqui estamos considerando textual tudo aquilo que é legível, por exemplo, através de uma ferramenta de edição de texto, seja um txt, html, csv, entre outros. Quando consideramos um arquivo como binário, significa que ele carrega outra codificação, seja uma estrutura de dados da linguagem ou até mesmo arquivos de imagem, vídeo, áudio, etc. Cada tipo de dado vai ter uma ou mais bibliotecas ou pacotes no Python capazes de interpretar sua codificação binária. Vamos conhecer o básico sobre manipulação de arquivos binários pois esse conhecimento será necessário quando falarmos de outras bibliotecas. O primeiro passo é relembrar que precisamos incluir explicitamente o modo de abertura "**b**" para indicar que o arquivo é binário.

```
f1 = open('binario.bin', 'wb')  # para escrita
f2 = open('binario.bin', 'r+b') # para leitura e escrita
f3 = open('binario.bin', 'ab')  # para adição
```

Como processar o conteúdo desses arquivos vai depender do tipo do dado. Falaremos caso a caso quando necessário. Vejamos por exemplo strings binárias. Note que usamos as mesmas funções **read** e **write**, porém precisamos do auxílio das funções de string **encode** e **decode** para as transformações de string para bytes e vice-versa.

```
frase = 'Olá mundo'
with open('texto.bin', 'wb') as fp:
fp.write(frase.encode()) # função encode codifica para bytes

f_reader = open('texto.bin', 'rb')
print(f_reader.read().decode()) # decodifica bytes para string
f_reader.close()
```

Algumas bibliotecas requerem uma variável manipuladora de arquivos para ler e interpretar o conteúdo. Por exemplo, a biblioteca Numpy para manipulação de arrays, que por hora vamos considerar como listas numéricas homogêneas.

```
import numpy as np
with open('lista.bin', 'rb') as fp:
lista = np.fromfile(fp, dtype=int)
print(lista)
# array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=np.uint8)
```

Para armazenar estruturas de dados do Python de forma genérica, contamos com pacotes como o **pickle**. Seu nome vem da palavra picles, que significa uma conserva de vegetais armazenada para consumo futuro. Suas funções para escrever ou ler arquivos binários, respectivamente **dump** e **load**, também requerem uma variável manipuladora de arquivos.

```
quadrados = [i**2 for i in range(6)]
with open('quadrados.bin', 'wb') as fp:
pickle.dump(quadrados, fp) # pickle.dump(dados, arquivo)

with open('quadrados.bin', 'rb') as fp:
dados = pickle.load(fp)    # dados = pickle.load(arquivo)
print(type(dados), dados) # <class 'list'> [0, 1, 4, 9, 16, 25]
```

Sua principal vantagem é ser capaz de armazenar qualquer tipo ou estrutura de dados do Python (ainda conheceremos várias!) de modo que ao lermos o arquivo já carregamos o dado em seu tipo original, com todas as funcionalidades de um objeto Python. No entanto, é exclusivo do Python. Não há garantias que os arquivos podem ser interpretados em outras linguagens. Você pode conhecer mais sobre o Pickle na documentação.

Outras estruturas

Outras estruturas

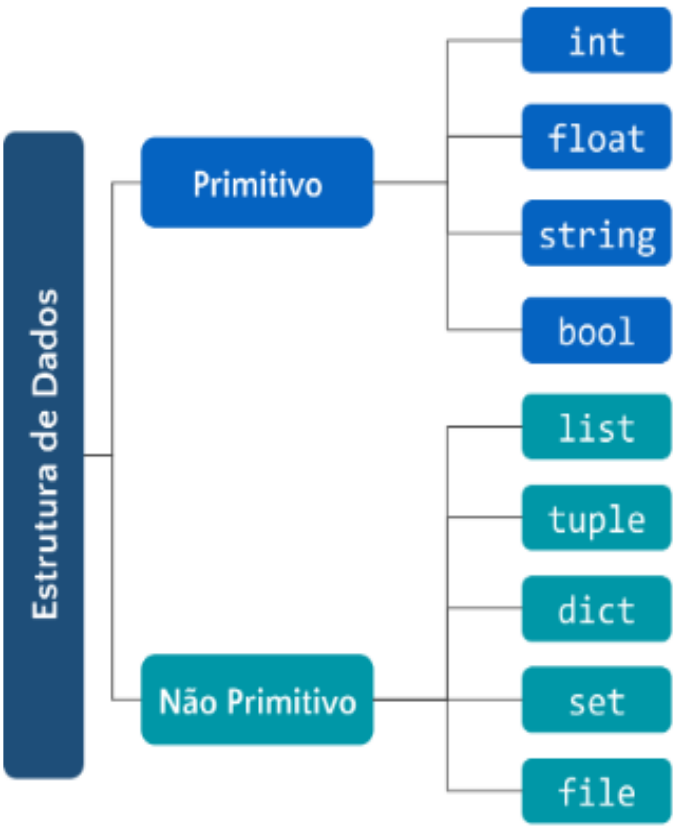
Todos os arquivos que não são textuais, são considerados binários. Mas é importante ressaltar que na prática todo arquivo é binário, inclusive textos. A extensão de um arquivo indica como seus bits serão interpretados. Ao abrir um arquivo no modo texto, estamos apenas indicando como interpretar as regras daquela codificação.

Neste módulo daremos continuidade às estruturas de dados não primitivas do Python. Agora que já conhecemos as listas, em toda a sua versatilidade, vamos introduzir outras 3 estruturas igualmente importantes:

•**set:** forma eficiente de manipular conjuntos de elementos únicos, permitindo operações de união, interseção, diferença, entre outros.

tupla: sequências imutáveis de elementos, permitindo representar dados heterogêneos e garantindo a integridade dos dados.

dicionário: estrutura de mapeamento que associa chaves a valores, proporcionando um acesso eficiente aos dados com base em uma chave única.



Set

O **set** em Python é uma implementação prática dos princípios estabelecidos pela teoria dos conjuntos, fortemente relacionados à álgebra booleana. "Set" em português significa conjunto, um conceito matemático que referencia uma **coleção de objetos distintos** sobre os quais podemos realizar operações de intersecção (**and**), união (**or**), diferença, entre outros.

As principais propriedades do set, que iremos expandir ao longo dessa aula, são:

- Não ordenado
- Heterogêneo
- Seus elementos devem ser de tipos hasheáveis
- Seus elementos são únicos, ou seja, duplicatas não são permitidas
- É um iterável mas não é indexável nem fatiável

Vamos começar pelas principais sintaxes para criação de um **set**. Apresentamos no código a seguir três diferentes maneiras:

•conversão de tipo de outro iterável	**s = set(iter)**
•sintaxe de chaves para fornecer elementos individuais	**s = {obj1,obj2,obj3}**
•criando um set vazio, o que só é possível com a palavra reservada set, já que a sintaxe de chaves vazias é reservada para outra estrutura, os dicionários	**s = set()**

```
>>> set([1,2,3])
{1, 2, 3}
>>> set("desenvolve")
{'v', 'e', 'l', 'd', 'o', 'n', 's'}
>>> s = {1,2,2,3,10}
>>> s
{3, 1, 10, 2}
>>> s = {"projeto", "desenvolve"}
>>> s
{'projeto', 'desenvolve'}
>>> s = set()
>>> s
set()
>>> type(s)
<class 'set'>
```

Note no exemplo acima mais uma expressão da dualidade de strings, que se convertida em set é considerada um iterável, e seus caracteres são adicionados ao set como elementos individuais. Por outro lado, podemos alimentar strings inteiras como elementos individuais de um set, e elas serão consideradas objetos atômicos imutáveis.

Isso nos leva à primeira importante propriedade dos sets: **seus elementos precisam ser *hasheáveis*.** Por simplicidade, é comum se referirem a essa propriedade como "elementos imutáveis", indicando que sets não aceitam objetos que podem sofrer alterações, como listas. Objetos imutáveis são como os tipos primitivos, cujo estado nunca é alterado. Já objetos hasheáveis permitem o cálculo de seu valor hash, uma espécie de assinatura digital. Se você consegue realizar a operação **hash(x)** com um objeto x, então x pode ser membro de um set. Se tentarmos atribuir um objeto não hasheável a um set, veremos o erro **unhashable type**.

```
>>> s = {3.14, [1,2,3]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Imutável vs Hasheável

Objetos imutáveis não podem ter seu estado alterado após criação, como os tipos primitivos **int, bool, float e string**. É importante não confundir variáveis com objetos. Criar uma variável é dar um nome a um objeto, o objeto é quem tem tipo e valor. Ao associar uma variável a um objeto do tipo inteiro, por exemplo, a variável pode ter seu valor alterado somente **se associada a um novo objeto**, o que implica no descarte do objeto anterior.

Vamos exemplificar com strings. Ao criar uma string, não podemos alterar o seu estado.

```
>>> nome = "Maria"
>>> nome[-1] = 'o'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> nome.append("s")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

A única maneira de "alterar" uma variável associada a uma string é substituir o valor dessa variável por um novo objeto, descartando o objeto anterior. A função **replace**, membro da string, não altera a string original, mas produz um novo objeto.

```
>>> nome.replace("a", "o")
Morio
>>> nome
Maria
>>> nome = nome.replace("a", "o")
```

Objetos hasheáveis, por outro lado, podem ser parâmetro da função **hash()**, a qual calcula um identificador numérico único para aquele elemento, como uma assinatura digital.

```
>>> hash("ola")
2181348669379246883
>>> hash([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Como o papel do hash é garantir a integridade de um elemento através de uma assinatura única, não faz sentido calculá-lo para elementos que podem sofrer alterações (o que produziria uma nova chave). Portanto, **todo objeto imutável é *hasheável***, mas o contrário não é verdade. O que define que um objeto é hasheável é possuir uma implementação para a função mágica **__hash__**. Futuramente aprenderemos que é possível criar tipos customizados de dados, que podem inclusive conter uma função **__hash__**.

A principal característica dos sets é não permitir duplicatas. Internamente os elementos de um set são armazenados de acordo com o seu valor hash, sua assinatura. Elementos duplicados possuem o mesmo valor **hash** e ocupariam a mesma posição na estrutura. Portanto, em um set serão descartadas quaisquer duplicatas, seja de um iterável convertido para set, ou novos elementos adicionados. Note que não há um erro associado à tentativa de adicionar uma duplicata, já que descartá-las é o comportamento esperado da estrutura.

```
>>> x = set(['ana', 'davi', 'ana', 'luiz'])
>>> x
{'luiz', 'ana', 'davi'}
>>> x.add('davi')
>>> x
{'luiz', 'ana', 'davi'}
```

Outra propriedade é que os elementos de um set **não são ordenados**. Isso implica que a estrutura não preserva a ordem de adição e não possui uma lógica facilmente previsível de ordenação. A ordem de armazenamento pode inclusive variar entre interpretadores.

No terminal	No Google Colab
`>>> set([1,2,20,1,6,6,210])` `{1, 2, 6, 210, 20}`	`1 set([1,2,20,1,6,6,210])` `{1, 2, 6, 20, 210}`

O **set também é uma estrutura heterogênea**, ou seja, comporta objetos de diferentes tipos na mesma estrutura. A única limitação é que todos os objetos sejam de algum tipo *hasheável*. A estrutura pode armazenar inclusive funções.

```
>>> s = set([2,'olá', False, 3.14])
>>> s
{False, 2, 3.14, 'olá'}
>>> def fn1(x): return x*100
>>> s.add(fn1)
>>> s
{False, 2, 3.14, 'olá', <function fn1 at 0x1024713a0>}
```

Por fim, mas não menos importante, **o set é um conjunto iterável, mas não é indexável nem fatiável**. Lembre-se que internamente ele não é organizado por ordem de adição, e sua ordenação pode variar de forma imprevisível, portanto o conceito de indexação como existe nas listas não faz sentido aqui.

```
>>> x = set(['ana', 'davi', 'ana', 'luiz'])
>>> for elemento in x:
...     print(elemento, end=" ")
...
luiz ana davi
>>> x[:2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

Operações e funções com set

Assim como nas listas, a expressão **elemento** in **conjunto** testa se um **elemento** pertence ao **conjunto**, retornando **True** caso o elemento esteja presente e **False** caso contrário. Lembre-se que a checagem pode ser negada com o uso do **not**.

```
>>> x = set([2,'olá', False, 3.14])
>>> 'olá' in x
True
>>> 4 not in x
True
```

Funções embutidas que trabalham com iteráveis também aceitam sets como parâmetros de entrada. Assim como em todos os casos, sua capacidade de aplicação depende do tipo dos dados membro da estrutura.

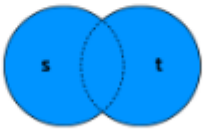
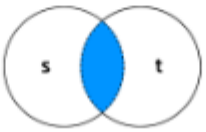
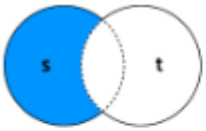
```
>>> x = {'ana', 'luiz', 'davi'}
>>> len(x)
3
>>> max(x)
'luiz'
>>> min(x)
'ana'
>>> sorted(x)
['ana', 'davi', 'luiz']
>>> print(sum(x))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Usando a função **dir**, conhecemos melhor as funções mágicas (**__func__**) e outras funções membro aplicáveis a um objeto do tipo set.

```
dir(set())
['__and__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__iand__', '__init__', '__init_subclass__', '__ior__', '__isub__', '__iter__', '__ixor__',
'__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__', '__reduce__',
'__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__setattr__', '__size-
of__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'clear', 'copy', 'dif-
ference', 'difference_update', 'discard', 'intersection', 'intersection_update',
'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference',
'symmetric_difference_update', 'union', 'update']
```

Quanto às funções mágicas, vale conhecê-las para inferir algumas informações sobre a estrutura, como o suporte a operadores relacionais (**__eq__: ==, __gt__: >, __le__: <=, __ne__: !=**), dentre muitos outros operadores e funções que você encontra na documentação.

Uma parcela dos operadores implementados para o set são simplificações das operações de conjunto fornecidas pela estrutura. Porém, enquanto as funções membro trabalham com qualquer iterável como parâmetro, os operadores trabalham apenas com objetos do tipo set. Note que as checagens de subset e superset retornam valores booleanos, enquanto as operações de conjunto união, intersecção, diferença e diferença simétrica retornam novos sets.

Função membro	Operador	Diagrama	Resultado
`s.issubset(t)`	`s <= t`		Testa se cada elemento de *s* está em *t*. Retorno booleano.
`s.issuperset(t)`	`s >= t`		Testa se cada elemento de *t* está em *s*. Retorno booleano.
`s.union(t)` *# or*	`s | t`		Retorna um novo set com o conjunto união de *t* e *s*
`s.intersection(t)` *# and*	`s & t`		Retorna um novo set com o conjunto intersecção de *t* e *s*
`s.difference(t)`	`s - t`		Retorna um novo set com elementos em *s* mas não em *t*
`s.symmetric_difference(t)`	`s ^ t`		Retorna um novo set com elementos em *s* ou *t* mas não em ambos

Algumas operações entre sets modificam o set que as chama, e não retornam nada. Essas são chamadas de operações *inplace* (no local).

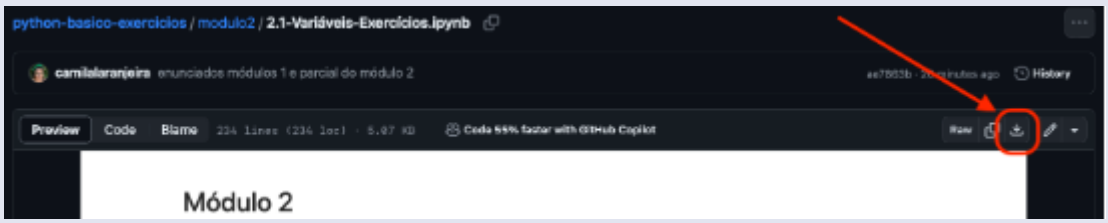
Função membro	Operador	Resultado
`s.update(t)`	`s |= t`	Atualiza *s* tornando-o conjunto união de *t* e *s*
`s.intersection_update(t)`	`s &= t`	Atualiza *s* tornando-o conjunto intersecção de *t* e *s*
`s.difference_update(t)`	`s -= t`	Atualiza *s* com os elementos em *s* mas não em *t*
`s.symmetric_difference_update(t)`	`s ^= t`	Atualiza *s* com elementos em *s* ou *t* mas não em ambos

E por fim, as funções a seguir também são realizadas inplace, no entanto operam sets com seus elementos (membros), além de não possuírem operadores equivalentes.

Função membro	Resultado
`s.add(x)`	Adiciona elemento imutável *x* ao set *s*
`s.remove(x)`	Remove elemento imutável *x* do set *s*; **`KeyError`** se elemento não encontrado.
`s.discard(x)`	Remove elemento imutável *x* do set *s* se estiver presente
`s.pop()`	Remove e retorna um elemento arbitrário do set *s*; **`KeyError`** se set vazio.

Exercício

1) Baixe o arquivo 8.1-Set_Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.



Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo8 do seu repositório.

Tupla

Com o nome em inglês tuple, é o tipo mais fundamental da linguagem. Suas principais propriedades são:

- Ordenado (por ordem de adição, assim como listas)
- Heterogêneo
- Iterável indexável e fatiável (somente para leitura, afinal é imutável)
- Imutável (principal propriedade! Mais eficiente e garante integridade)
    - Seus elementos, no entanto, não precisam ser imutáveis

A seguir apresentamos as maneiras de criar uma tupla.

•conversão de tipo de outro iterável	t = tuple(iter)
•sintaxe de parênteses para fornecer elementos individuais	t = (obj1,obj2,obj3)
•criando uma tupla vazia, com a palavra reservada tuple	t = tuple()
•criando uma tupla vazia, com a sintaxe de parênteses	t = ()

```
>>> tuple(range(1, 4))
(1, 2, 3)
>>> t = (1,2,3)
>>> t
(1, 2, 3)
>>> type(t)
<class 'tuple'>
```

Como já estamos bem familiares com o significado das propriedades de estruturas de dados, o código a seguir vai exemplificar simultaneamente as características da tupla: ordenada, heterogênea, iterável indexável e fatiável e imutável.

```
>>> data_nasc = (8, 2, 93)
>>> dados = tuple(["Camila", "Laranjeira", data_nasc]) # heterogênea
>>> for dado in dados: print(dado, end = " ")  # ordenada e iterável
Camila Laranjeira (8, 2, 93)
>>> dados[0]  # indexação
'Camila'
>>> dados[:2] # fatiamento
('Camila', 'Laranjeira')
>>> dados[0] = 'Camilla'  # imutabilidade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

O detalhe importante é que a tupla como estrutura não pode ser alterada, no sentido que seus elementos não podem ser deletados ou novos elementos adicionados. Note a seguir que a tupla não possui funções de adição ou remoção. Além das funções mágicas, seus únicos membros são as funções **count(x)** e **index(x)** que respectivamente contam a quantidade de um dado elemento **x** e localiza o índice da primeira ocorrência de **x** na estrutura.

```
>>> dir(dados)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__
dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__geti-
tem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__
subclasshook__', 'count', 'index']
```

No entanto, a tupla aceita objetos de qualquer tipo, inclusive tipos mutáveis como listas.

```
>>> triangulo = ([0,0], [3, 2], [-1, 4])
>>> triangulo[0] = [1,1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> triangulo[0][0], triangulo[0][1] = 1, 1
>>> triangulo
([1, 1], [3, 2], [-1, 4])
```

Operações e funções com tuplas

Quando fazemos atribuições múltiplas ou retornamos múltiplos valores em funções, o Python automaticamente empacota esses dados como uma tupla. Ou seja, parênteses são opcionais na criação de tuplas. Quaisquer sequência de valores separado por vírgula, se atribuída a uma única variável, é empacotada. Até mesmo valores únicos.

```
>>> frutas = 'maçã', 'uva', 'banana'
>>> type(frutas)
<class 'tuple'>
>>> legume = 'berinjela',
>>> type(legumes)
<class 'tuple'>
```

```
def novo_cadastro():
        usuario = input()
senha   = input()
return usuario, senha

cadastro = novo_cadastro()
print(type(cadastro))
# <class 'tuple'>
```

Assim como múltiplos elementos são implicitamente empacotados em uma tupla, também podemos **desempacotar uma tupla** em múltiplas variáveis. Essa operação funciona com outros iteráveis, como listas, mas é comum em tuplas pela sua natureza imutável.

```
>>> cadastro
('camila', '123456')
>>> usuario, senha = cadastro
>>> usuario
'camila'
>>> senha
'123456'
```

```
>>> cadastro = list(cadastro)
>>> cadastro
['camila', '123456']
>>> usuario, senha = cadastro
>>> usuario
'camila'
>>> senha
'123456'
```

Python também possui o operador unário * para empacotar ou desempacotar um subconjunto do iterável de forma flexível. Ao preceder uma variável do lado esquerdo de uma atribuição, esse operador indica que a variável será associada a um subconjunto dos elementos do lado direito da atribuição. Esse subconjunto é associado como um objeto do tipo lista.

```
>>> frutas = 'uva', 'maçã', 'umbu', 'caju'
>>> primeiro, *outros = frutas
>>> outros
['maçã', 'umbu', 'caju']
>>> *outros, ultimo = frutas
>>> outros
['uva', 'maçã', 'umbu']
>>> primeiro, *outros, ultimo = frutas
>>> outros
['maçã', 'umbu']
```

Assim como as listas, tuplas implementam os operadores aritméticos da soma (**+**) e da multiplicação (*), respectivamente concatenação e repetição.

```
>>> n1, n2 = (1,2,3), (4,5,6)
>>> n1 + n2
(1,2,3,4,5,6)
>>> n1 * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Assim como os outros iteráveis que conhecemos, podemos checar se um elemento faz parte da tupla usando o operador **in**, ou sua negação **not in**.

```
>>> x = tuple(i**2 for i in range(4))
>>> x
(0, 1, 4, 9)
>>> 0 in x
True
>>> 5 not in x
True
```

Comparações entre tuplas podem ser feitas com operadores relacionais. Eles operam elemento a elemento, comparando elementos em índices equivalentes. Lembrando que operadores relacionais não são aplicáveis a quaisquer combinações de tipos.

```
>>> (5,6,7) < (7,5) # True na primeira comparação
True
>>> (5,2,7) > (5,6) # False na segunda comparação
False
>>> (4,3,2) <= (4,3,2) # Realiza todas as comparações
True
>>> (2,'ana',11) < (5,5) # True na primeira comparação
True
>>> (5,'ana',11) < (5,5) # Falha na segunda comparação
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

Enumerate

Uma função muito útil para laços é a **enumerate**. Ela adiciona um contador a um dado iterável, iterando simultaneamente no contador e nos valores do iterável. A cada iteração retorna a tupla (contador, valor). Os códigos a seguir apresentam o **enumerate** e sua equivalência se criássemos manualmente o mesmo tipo de contador.

```
cont = 0
nums = [13, 25, 18, 11]
for valor in nums:
    print(cont, valor)
    cont += 1
```

```
0 13
1 25
2 18
3 11
```

```
nums = [13, 25, 18, 11]
for tup in enumerate(nums):
    cont, valor = tup
    print(cont, n)
```

```
0 13
1 25
2 18
3 11
```

Vale ressaltar que o retorno de **enumerate** é um **objeto gerador**, portanto não está sujeito a indexação ou fatiamento. O papel da função é ser utilizada em laços de repetição. Diferente do que fizemos no exemplo acima, o seu uso mais comum já desempacota os elementos do enumerate na própria instrução do **for**. Vejamos a seguir um exemplo mais complexo.

```
dias = [('jan', 31),  ('fev', 29), ('dez',31)]
for indice, elemento in enumerate(dias):
        print(indice, '-', elemento)
```

```
0 - ('jan', 31)
1 - ('fev', 29)
2 - ('dez', 31)
```

```
dias = [('jan', 31),  ('fev', 29), ('dez',31)]
for indice, (mes, dia) in enumerate(dias):
        print(indice,'-', mes, '-', dia)
```

```
0 - jan - 31
1 - fev - 29
2 - dez - 31
```

Como os elementos do iterável são iteráveis, vemos dois níveis de desempacotamento. O primeiro nível **(indice, elemento)** se refere à tupla produzida pelo enumerate. Já no segundo, cientes que cada elemento da lista é uma tupla, podemos desempacotá-los em variáveis distintas **(indice, (mes, dia))**.

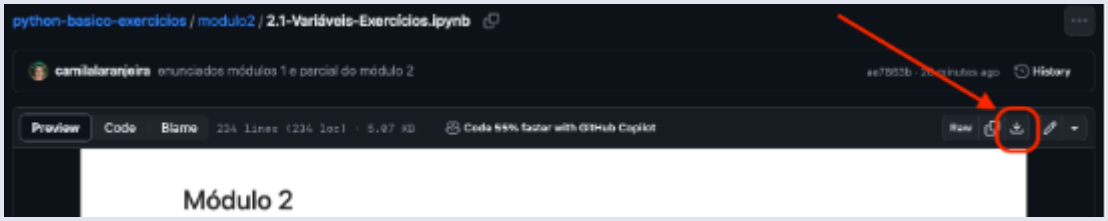
Por padrão, a contagem se inicia em zero, mas podemos alterar essa informação através do argumento **start** da função. Ele aceita um valor inteiro qualquer e inicia a contagem a partir daquele ponto.

```
for cont, val in enumerate(range(3), start=-2):
        print(cont, val)
```

```
-2 0
-1 1
0 2
```

Exercício

1) Baixe o [arquivo 8.2-Tupla Exercícios.ipynb]() como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.



Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo8 do seu repositório.

Dicionários

Enquanto outras estruturas, como listas e tuplas, são acessíveis pela sua posição **(índice)**, o dicionário introduz o conceito de itens rotulados **(chave)**.

Índice: "O segundo pote à esquerda"

Chave: "O pote escrito macarrão"

Conhecido como estrutura associativa, dicionários armazenam pares **(chave, valor)**, associando chaves de acesso ao conteúdo. Em outras linguagens essa mesma estrutura pode ser chamada de array associativo (Perl / PHP), properties (C# / .Net), map ou hashmap (Java).

A seguir apresentamos as maneiras de criar uma tupla. Note que usamos uma lista de tuplas como o iterável de exemplo para criação através de conversão. A tupla será um forte parceiro dos dicionários ao longo de toda essa aula.

•conversão de tipo de outro iterável	d = dict([(<chave>, <valor>), ...])
•sintaxe de chaves e dois pontos	d = {c1: v1, c2: v2, c3: v3}
•criando um dicionário vazio	d = dict() d = {}

Note que a sintaxe de chaves também é usada para os sets, onde cada elemento é um único objeto → **s = {e1, e2, e3, ..., en}.** Já o dicionário depende de pares de objetos **{<chave>: <valor>}**. Há um caso especial de definição, quando as chaves são sempre strings. Podemos definir o dicionário como parâmetros nomeados com a palavra reservada dict.

d = dict(c1=v1, c2=v2, c3=v3)

Vejamos na prática todas essas maneiras de criar dicionários.

```
>>> dict([('Acre','Rio Branco'), ('Amapá', 'Macapá')])
{'Acre': 'Rio Branco', 'Amapá': 'Macapá'}

>>> d = {'Acre': 'Rio Branco', 'Amapá': 'Macapá'}
>>> d
{'Acre': 'Rio Branco', 'Amapá': 'Macapá'}

>>> capitais = dict(Acre="Rio Branco",
...             Amapa="Macapá",
...             Amazonas="Manaus")
```

Podemos comparar o uso de dicionários ao de listas, comparando o acesso por índice com o acesso por chave. Note que em ambos os casos usamos a sintaxe de colchetes após o nome da variável para indicar qual elemento desejamos acessar.

```
# lista
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> lst
[21, 183]
>>> lst[0] = 23
>>> lst
[23, 183]
```

```
# dicionário
>>> ddd = dict() # ddd = {}
>>> ddd['idade'] = 21
>>> ddd['curso'] = 182
>>> ddd
{'curso': 182, 'idade': 21}
>>> ddd['idade'] = 23
>>> ddd
{'curso': 182, 'idade': 23}
```

Ao criar um dicionário, usamos a chave como referência de acesso. Assim como índices são únicos em listas, aqui **chaves são únicas**. Atribuir um novo valor a uma chave existente apenas atualiza o valor. Para adicionar um novo elemento a um dicionário, basta fazer uma atribuição usando uma chave que ainda não exista naquela estrutura.

```
>>> capitais = dict(Acre="Rio Branco", Amapa="Macapá")
>>> capitais["Acre"] = "RBR"        # sobrescrevendo elemento existente
>>> capitais
{'Acre': 'RBR', 'Amapá': 'Macapá'}
>>> capitais['Amazonas'] = 'Manaus' # adicionando novo elemento
```

Acessar chaves inexistentes lança um KeyError ou erro de chave. A função membro **get** permite definir um retorno padrão para tratar esses casos.

```
>>> capitais = dict(Acre="Rio Branco", Amapa="Macapá")
>>> capitais["Amazonas"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  KeyError: 'Amazonas'
>>> capitais.get("Amazonas", False)
False
```

Quem pode ser chave?

No par **<chave>: <valor>**, o valor pode ser de qualquer tipo válido da linguagem, inclusive outros dicionários. Valores podem se repetir e podem ser mutáveis. Sem restrições! Já as **chaves precisam ser de um tipo hasheável** (vimos esse conceito na

aula de sets). A estrutura interna dos dicionários depende da chave para encontrar elementos. Ela precisa ser consistente ao longo da execução. Vale lembrar que a tupla é um objeto imutável e portanto também pode servir como chave.

```
d1 = {1: 'Jan', 2: 'Fev'}            # chave numérica
d2 = {'Jan': {'id': '01', 'dias': 31},  # chave string
      'Fev': {'id': '02', 'dias': 28}}
d3 = {('Jan', 1): 31, ('Fev', 2): 28}   # chave tupla
```

Operações e funções com dicionários

Dicionários são iteráveis, porém nativamente itera nas chaves. Podemos usar as funções membro **items**, **keys** ou **values** para explícitamente escolher onde queremos iterar.

•**items:** retorna um iterável de tuplas com os pares (chave, valor)

•**keys:** retorna um iterável com as chaves

•**values:** retorna um iterável com os valores

```
>>> frutas = {'uva': 8.50, 'umbu': 34.0, 'morango': 21.99}
>>> for f in frutas: print(f, end=" ")
uva umbu morango
>>> for key in frutas.keys(): print(key, end=" ")
uva umbu morango
>>> for value in frutas.values(): print(value, end=" ")
8.5 34.0 21.99
>>> for key, value in frutas.items(): print(key, value, end=" ")
uva 8.5 umbu 34.0 morango 21.99
```

Se aplicarmos diretamente em um dicionário as funções embutidas que aceitam iteráveis, é preciso estar ciente que elas estarão atuando sobre as chaves.

```
>>> frutas = {'uva': 8.50, 'umbu': 34.0, 'morango': 21.99}
>>> max(frutas)
'uva'
>>> sorted(frutas)
['morango', 'umbu', 'uva']
>>> sum(frutas)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Assim como listas, também podemos construir expressões para inicialização rápida de dicionários. Recebendo o nome de compreensão de dicionários (dict comprehension), usa a sintaxe de chaves e dois pontos.

```
# lista
>>> [(x, x**2) for x in range(4)]
[(0, 0), (1, 1), (2, 4), (3, 9)]
```

```
# dicionário
>>> {x: x**2 for x in range(4)}
{0: 0, 1: 1, 2: 4, 3: 9}
```

A compreensão de dicionários possui o mesmo gtrau de liberdade de listas, permitindo a inclusão de expressões condicionais de **if** ou **if-else**.

```
>>> dic = {x: 1/x if x != 0 else 0 for x in range(-5, 5)}
>>> dic
{-5: -0.2, -4: -0.25, -3: -0.3333333333333333, -2: -0.5, -1: -1.0, 0: 0,  1: 1.0, 2:
0.5, 3: 0.3333333333333333, 4: 0.25}
```

Além das funções que já conhecemos, lembre-se que você pode usar a função embutida dir e consultar a documentação para conhecer as outras funções de dicionário.

```
>>> dir(dict)
['__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__ge-
titem__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__ior__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__re-
duce__', '__reduce_ex__', '__repr__', '__reversed__', '__ror__', '__setattr__', '__
setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromke-
ys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

sorted(..., key=...)

A palavra key (chave) é também usada no Python para se referir a um interessante parâmetro das funções de ordenação de iteráveis. Por exemplo, se queremos ordenar o dicionário de frutas pelos seus preços, ou seja, pelos valores e não pelas chaves, isso não é diretamente alcançável aplicando a função **sorted**. Lembre-se que dicionários tendem a sempre se referir às suas chaves.

Uma maneira simples é adicionar uma "chave de ordenação" à função **sorted**. Em outras palavras, criamos uma função que será executada para cada elemento do iterável que queremos ordenar. O papel dessa função é ditar qual transformação devemos aplicar ao elemento. A ordenação será feita de acordo com os retornos dessa função. A função fornecida pode ser customizada, embutida, ou até mesmo uma lambda, como no exemplo a seguir.

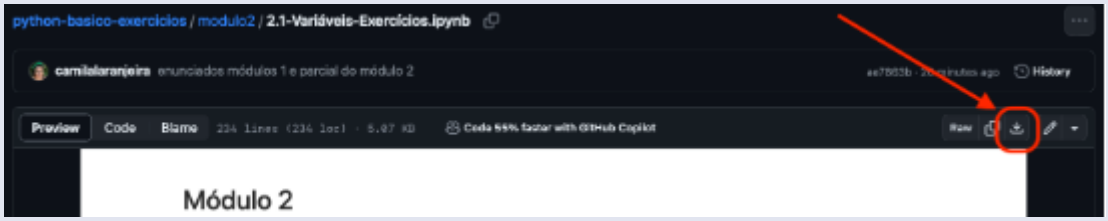
```
>>> frutas = {'uva': 8.50, 'umbu': 34.0, 'morango': 21.99}
>>> sorted(frutas.items()) # ordenação padrão pela chave
[('morango', 21.99), ('umbu', 34.0), ('uva', 8.5)]
>>> sorted(frutas.items(), key = lambda x: x[1])
[('uva', 8.5), ('morango', 21.99), ('umbu', 34.0)]
```

Esse artifício é aplicável a outros iteráveis. Não confunda as chaves! Note que a função retorna os elementos originais. As operações da função são aplicadas apenas à regra de ordenação. Você também pode consultar a documentação para saber mais sobre o **sorted** e suas chaves de ordenação.

```
>>> sorted("O rato roeu a roupa do Robson".split())
['O', 'Robson', 'a', 'do', 'rato', 'roeu', 'roupa']
>>> sorted("O rato roeu a roupa do Robson".split(), key=str.lower)
['a', 'do', 'O', 'rato', 'Robson', 'roeu', 'roupa']
```

Exercício

1) Baixe o arquivo 8.3-Dicionario_Exercícios.ipynb como indicado na figura abaixo, abra para edição no Jupyter Notebook, e preencha as respostas das atividades.



Instruções para submissão: envie o arquivo acima preenchido com as suas respostas para a pasta modulo8 do seu repositório.

PROJETO
DESENVOLVE