



LÓGICA DE PROGRAMAÇÃO E ALGORITMOS

AULA 2

CONVERSA INICIAL

O objetivo desta aula é darmos nossos primeiros passos com a linguagem Python. Iremos colocar os dedos no teclado e começar a desenvolver nossos primeiros algoritmos, enquanto aprendemos de maneira dinâmica os primeiros recursos dessa poderosa linguagem de programação.

Ao longo desta aula, você vai aprender como é o processo de execução de um algoritmo computacional. Aprenderá a gerar entrada e saída de dados no programa, bem como manipular dados e variáveis ao longo do processamento do algoritmo pelo computador.

Todos os exemplos apresentados neste material poderão ser praticados concomitantemente em um *Jupyter Notebook*, como o *Google Colab* (mais detalhe a seguir), e não requer a instalação de nenhum software de interpretação para a linguagem Python em sua máquina.

Ao final do material, você encontrará alguns exercícios resolvidos, que estão colocados em linguagem Python, pseudocódigo e também em fluxograma.

TEMA 1 – AMBIENTES DE DESENVOLVIMENTO

Você já conheceu a linguagem Python, sua história e as características que a tornaram bastante popular. Vamos agora para o primeiro assunto desta aula, em que aprenderemos sobre as ferramentas de desenvolvimento para Python.

A linguagem Python acompanha, por padrão, um interpretador denominado de IDLE. O interpretador é um software que aceita comandos escritos em Python. Diferente de um compilador, ele executa linha por linha sem gerar um código de máquina de todo o programa. O IDLE é um ambiente de desenvolvimento integrado e que existe para todos os sistemas operacionais.

Atualmente o Python encontra-se na versão 3 (a *release* irá depender de quando este material

estiver sendo lido). No mês de março de 2020, o Python encontra-se na versão 3.8.2. Precisamos do interpretador da linguagem para esta versão.

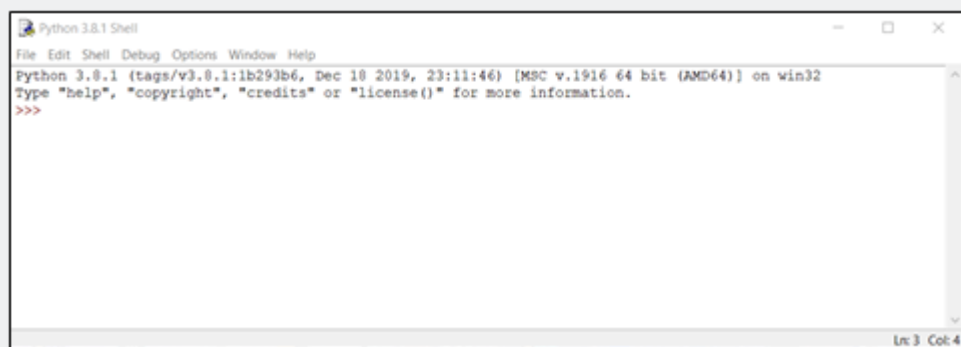
Saiba mais

Em ambientes Windows, o Python não vem instalado por padrão no sistema operacional, sendo necessário fazer seu download e seguir os passos de instalação

PYTHON. Download. **Python**, [S.d.]. Disponível em: [<https://www.python.org/downloads/>](https://www.python.org/downloads/). Acesso em: 6 set. 2020.

O processo de instalação do Python em Windows não será elaborado neste documento, uma vez que ele poderá sofrer modificações com o decorrer do tempo. Nesta disciplina, você encontrará um tutorial adicional dentro do AVA que conterà o processo de instalação para Windows. Ele será atualizado sempre que o Python também sofrer alteração na sua instalação. Caso esteja trabalhando em Linux, praticamente todas as distribuições já vêm com o Python e o IDLE instalados.

Figura 1 – IDLE Python Shell 3.8 em ambiente Windows



A linguagem Python, assim como qualquer outra linguagem, também contém ambientes de desenvolvimento integrados com interfaces gráficas (*Integrated Development Environment – IDE*).

Saiba mais

Um dos mais conhecidos, aberto para a comunidade, é o *PyCharm*, desenvolvimento pela

empresa tcheca JetBrains:

PYCHARM. Disponível em: <https://www.jetbrains.com/pt-br/pycharm/>. Acesso em: 6 set. 2020.

Nesse tipo de ferramenta, conseguimos desenvolver códigos em linguagem Python. Estas ferramentas são mais empregadas no âmbito profissional (ao invés do IDLE), pois realizam testes mais completos, que envolvem depuração e execução passo a passo do seu código. A ferramenta *PyCharm* será mais bem apresentada posteriormente, em outra oportunidade.

Por fim, temos hoje uma nova maneira de desenvolvermos em Python, o Projeto Jupyter (*Jupyter Project*), que é uma plataforma criada com fins não lucrativos e feita para o desenvolvimento de *softwares open-sources*. O Projeto Jupyter é uma plataforma capaz de executar na nuvem códigos em linguagem Python. Desse modo, você não necessita instalar nada na sua máquina, bastando acessar a plataforma e criar um *Notebook Jupyter* (aqui o termo *notebook* refere-se ao termo em inglês *bloco de anotações*, e não a um computador portátil, que no Brasil chamamos de *notebook* ou *laptop*). Ao fazer isso, você ganha um espaço de memória, bem como uma capacidade de processamento, em um servidor para executar seus programas em Python. Desse modo, todo o processamento é feito em um servidor, e sua máquina só precisa ter uma conexão estável com a internet.

Os *Notebooks Jupyter* são também ótimas ferramentas didáticas, pois podem misturar textos e códigos interativos dentro de um mesmo bloco de anotações. Sendo assim, vamos adotar esta ferramenta para realizarmos nossos estudos em Python ao longo de nossas aulas teóricas.

Saiba mais

Existem diferentes plataformas que dão suporte ao desenvolvimento em *Notebooks Jupyter*. Dentre as mais populares, estão:

1. Microsoft Azure Notebook. Disponível em: <https://notebooks.azure.com/>. Acesso em: 6 set. 2020.

2. Google Colab – Colaboraty. Disponível em: <https://colab.research.google.com/notebo>

Vamos adotar a plataforma do Google pela facilidade de integração com as ferramentas do Google. Inclusive, todo este material que você está lendo também está colocado de maneira interativa dentro do *Google Colab*. No entanto, caso tenha interesse em utilizar outro *Notebooks Jupyter*, saiba que é possível abrir o mesmo documento em diferentes ferramentas, pois são compatíveis entre si.

TEMA 2 – CICLO DE PROCESSAMENTO DE DADOS

Estamos praticamente prontos para iniciarmos nossos primeiros programas em linguagem Python. Conforme vimos anteriormente, um algoritmo é um conjunto bem definido de instruções executadas sequencialmente. Todo e qualquer programa computacional é um algoritmo e contempla os três blocos apresentados a seguir:

- Entrada – maneira como as informações são inseridas no programa. Lembra o nosso exemplo do sanduíche, visto anteriormente? Os ingredientes são nossos dados de entrada, os quais serão usados nas manipulações futuramente. A entrada padrão adotada em programas computacionais é a inserção de dados via teclado, porém, é possível que os dados venham ao programa de outra maneira, como conexão de rede, ou de outro programa executando na própria máquina;
- Processamento – representa a execução das instruções, envolvendo cálculos aritméticos, lógicos, alterações de dados etc. Em suma, tudo o que é executado de alguma maneira pela CPU e gravado ou buscado na memória;
- Saída – após o processamento das instruções, é necessário que o resultado do nosso programa seja disponibilizado ao usuário de alguma maneira. Lembra-se do nosso exemplo do sanduíche que já utilizamos em outra oportunidade? O sanduíche montado no final seria nossa saída, pronto para ser degustado. A saída padrão adotada em um programa computacional é uma tela/*display*, onde as informações podem ser exibidas. No entanto, é possível que nossa saída seja o envio de dados via conexão de rede, ou a impressão de um documento na impressora. Se estivéssemos desenvolvendo um algoritmo para uma *Raspberry Pi* ou um *Arduino*, a saída poderia ser uma luz acendendo ou piscando, por

exemplo.

O fluxo de execução de um algoritmo sempre se dá conforme a Figura 2, da esquerda para a direita. Primeiramente, obtemos os dados de entrada do programa, os quais são usados ao longo de toda a execução do algoritmo. Em seguida, com os dados obtidos, processamos no hardware da máquina, gerando uma saída, normalmente em um monitor/tela.

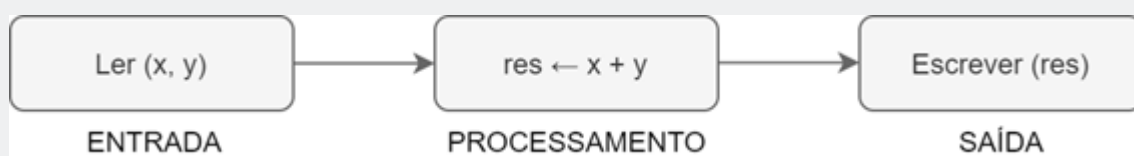
Figura 2 – Ciclo de processamento de dados



Vejamos um exemplo prático de ciclo de processamento para um programa que soma dois valores e apresenta na tela o resultado dessa soma. No bloco de entrada, temos a leitura desses valores via teclado, por exemplo. Nós os chamamos aqui de x e y .

Em seguida, o processamento da máquina se encarrega de calcular isso, buscando os dados na memória e operando-os dentro da CPU. O resultado final é gravado novamente na memória do programa, podendo ser impresso na tela (bloco de saída).

Figura 3 – Exemplo de uma soma de dois valores – ciclo de processamento de dados

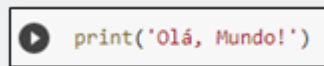


2.1 O PRIMEIRO PROGRAMA

Chegou o grande momento! Agora que sabe o ciclo de funcionamento todo algoritmo computacional, você irá desenvolver seu primeiro programa em Python. Existe uma tradição na área da computação que diz que sempre devemos iniciar nosso primeiro programa computacional imprimindo na tela uma mensagem que diz “Olá, Mundo!”. Se você mantiver a tradição, conta a lenda que será um ótimo programador. Então, não pode quebrar essa tradição, não é mesmo? Vamos fazer isso!

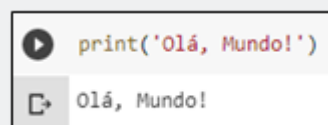
Vale ressaltar aqui que você pode testar os códigos em uma IDE de sua preferência, como o IDLE ou o *PyCharm*, porém, recomendamos que, para fins didáticos, trabalhe no *Google Colab*, no qual todos os exemplos do nosso material foram testados. Digite o seguinte comando na ferramenta:

Figura 4 – Desenvolvendo o primeiro programa 1



Veja que do lado da instrução que você digitou existe um ícone de uma seta para a direita, *Executar célula*. Ao clicar neste ícone, você verá que, após alguns segundos, seu comando será executado. Irá aparecer uma nova linha abaixo da que você digitou, com o resultado do seu comando. Veja:

Figura 5 – Desenvolvendo o primeiro programa 2



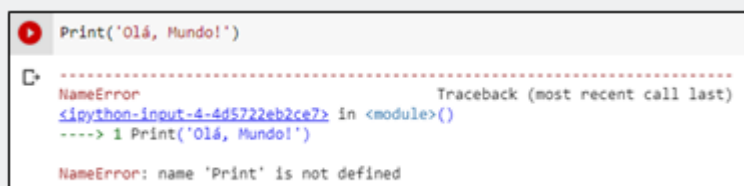
O resultado apresentado é a saída do programa, gerada com base na instrução que você digitou. Essa instrução serve para mostrar na tela do computador uma mensagem (mais detalhes sobre ela em seguida). É o que, literalmente, aparecerá na tela para um usuário, caso esse usuário estivesse executando seu programa. Ele receberia na tela a mensagem de *Olá, Mundo!*

Agora que você já manteve a tradição e executou seu primeiro programa, aqui vão mais algumas dicas e regrinhas básicas:

- Sempre verifique (duas, três, cinco, dez vezes!) se você digitou corretamente cada letra, número, ou caractere especial, caso contrário, erros poderão aparecer na tela;
- Atenção! Letras maiúsculas e minúsculas são compreendidas de maneira completamente diferentes pelo Python. Por exemplo, escrever *print* e escrever *Print* são coisas distintas.

Neste caso, se utilizar a letra maiúscula *P*, veja o que acontece caso tentemos executar no *Google Colab*. Um erro aparece informando que *Print* não foi encontrado/definido (*NameError: name 'Print' is not defined*).

Figura 6 – Letras maiúsculas e minúsculas



```
Print('Olá, Mundo!')
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-4d5722eb2ce7> in <module>()
----> 1 Print('Olá, Mundo!')
NameError: name 'Print' is not defined
```

- Sempre que você abrir aspas, lembre-se de fechá-las. Se existe uma aspa, existirá uma segunda para funcionar como delimitador. O mesmo vale para parênteses;
- Atenção aos espaços! O Python usa espaços em branco para uma série de operações, as quais iremos trabalhar no futuro. Sendo assim, sempre digite seus programas usando o mesmo alinhamento apresentado neste material ou nos livros didáticos de Python que estiver estudando.

2.2 FUNÇÃO DE SAÍDA

Vamos analisar nosso primeiro programa de uma maneira mais detalhada. O *print* pode ser chamado de *comando* ou *instrução*, porém, o termo mais adequado seria *função*.

Entenda por ora que uma função é um código previamente desenvolvido dentro da linguagem e responsável por executar determinada ação. Nesse caso, a ação tomada pelo *print* é a de exibir na tela do computador a mensagem que estiver dentro dos parênteses.

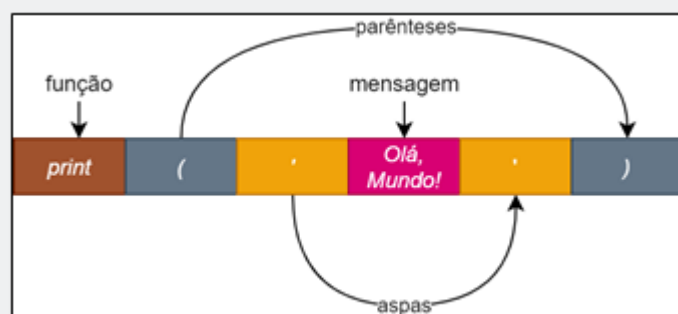
Sempre que possível, iremos aprender o equivalente em pseudocódigo da função, ou recurso, que acabamos de aprender. O equivalente ao *print* em pseudocódigo é o comando *Escrever*, lembrando que, conforme você já sabe, o português estruturado é uma representação que independe de uma linguagem.

Figura 7 – Função de saída em pseudocódigo e em linguagem Python



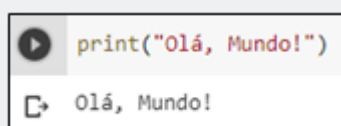
Podemos usar o *print* para mostrar uma mensagem de texto, como fizemos em nosso primeiro programa. Para que possamos mostrar um texto literal na tela, precisamos delimitá-lo por aspas. Observe essa estrutura completa na Figura 8.

Figura 8 – Detalhando a função *print* para escrita de mensagens de texto



Embora a comunidade adote amplamente as aspas simples, a linguagem Python também aceita o uso de aspas duplas. Veja:

Figura 9 – Aspas duplas



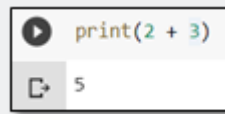
Saiba mais

Experimente agora executar a função *print* para escrever uma mensagem qualquer na tela, mas agora sem usar nenhum tipo de aspas. O que aconteceu? Por quê?

O *print* também pode ser utilizado para realizar operações aritméticas. Nesse caso, iremos trabalhar sem o uso das aspas. Vamos realizar a operação de $2 + 3$. O resultado esperado dessa

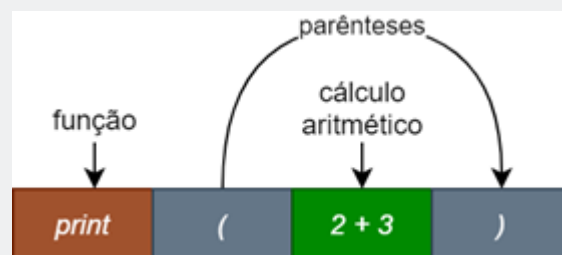
adição é 5, vejamos:

Figura 10 – Resultado da operação



Observe que, ao removermos as aspas, o próprio *Google Colab* alterou a cor do que está dentro dos parênteses para verde. Isso significa que não temos mais um texto ali dentro, mas sim uma operação aritmética que deve ser resolvida.

Figura 11 – Detalhando a função *print* para escrita de operações aritméticas



Vamos experimentar alguns testes diferentes. O que acontecerá se eu executar $2 + 3$ entre aspas? Lembre-se: o uso das aspas indica um texto, literalmente, sendo colocado na tela. Portanto, tudo dentro das aspas aparece como mensagem. Assim, a mensagem $2 + 3$ é colocada na tela:

Figura 12 – Mensagem $2 + 3$



Continuando nossos testes, é possível colocarmos somente os números entre aspas, deixando o operador de adição fora delas. O resultado disso será a junção (concatenação) de suas mensagens. Neste caso, as mensagens são dois números. Desse modo, o número 2 será

colocado seguido pelo número 3 na tela, ficando 23:

Figura 13 – Mensagem 23

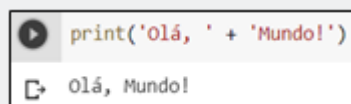


```
print('2' + '3')
```

23

Podemos escrever também nossa mesma frase anterior (*Olá, Mundo!*), mas agora concatenando duas mensagens, e o resultado é o mesmo na tela:

Figura 14 – Mensagem *Olá, Mundo!*

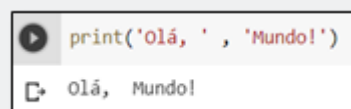


```
print('Olá, ' + 'Mundo!')
```

Olá, Mundo!

Em vez de usar o sinal de adição, a concatenação pode ser feita utilizando uma vírgula. Mas atenção: não confunda a virgula da concatenação com a virgula colocada dentro da mensagem:

Figura 15 - Concatenação



```
print('Olá, ', 'Mundo!')
```

Olá, Mundo!

Por fim, podemos misturar no mesmo *print* uma mensagem concatenada com uma operação aritmética. Nesse caso, faz-se necessário concatenar com a vírgula, obrigatoriamente. Percebam que o que ficou entre aspas é colocado literalmente na tela, já o que está fora das aspas é calculado:

Figura 16 – Mensagem concatenada com uma operação aritmética



```
print('O resultado da soma de 2 + 3 é: ', 2 + 3)
```

O resultado da soma de 2 + 3 é: 5

2.3 OPERADORES E OPERAÇÕES MATEMÁTICAS

Vimos que podemos realizar operações aritméticas utilizando a função *print*. Assim, quais operadores aritméticos básicos temos na linguagem Python? A Tabela 1 apresenta a lista desses operadores.

Tabela 1 – Lista de operadores matemáticos em Python e em pseudocódigo

Pseudocódigo	Python	Operação
+	+	Adição
-	-	Subtração
*	*	Multiplicação
/	/	Divisão (com casa decimais)
Não existe	//	Divisão (somente a parte inteira)
MOD	%	Módulo/resto da divisão
^	**	Exponenciação ou potenciação

Os cálculos de expressões aritméticas funcionam da mesma maneira que na matemática tradicional. Ou seja, a ordem de precedência dos operadores deve ser respeitada da mesma maneira que você respeita ao fazer um cálculo na ponta do lápis. Vejamos a equação a seguir:

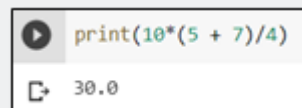
$$10 \times \left(\frac{5 + 7}{4} \right)$$

Na matemática, multiplicação e divisão acontece antes de uma adição, porém os cálculos dentro dos parênteses têm uma prioridade ainda maior. Sendo assim, a multiplicação acontecerá

no final.

A equação matemática está construída em Python logo a seguir. Observe que temos um parêntese que envolve o 5 e o 7, indicando que primeiro a adição acontecerá, para depois a multiplicação ocorrer e, por fim, a divisão.

Figura 17 -Equação em Python



```
print(10*(5 + 7)/4)
```

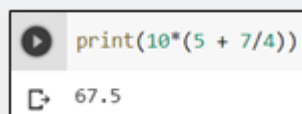
30.0

Saiba mais

Experimente agora trocar o símbolo da divisão da equação pelas duas barras //. O que aconteceu com o resultado? Por quê?

Observe que, se colocarmos os parênteses envolvendo também o número 4, o resultado ficará diferente, e incorreto, pois não condiz mais com a expressão aritmética inicial.

Figura 18 – Resultado incorreto



```
print(10*(5 + 7/4))
```

67.5

A expressão que corresponde ao código acima é a colocada a seguir. Percebeu a diferença que isso dá no resultado?

$$10 \times \left(5 + \frac{7}{4} \right)$$

Saiba mais

Escreva as seguintes expressões matemáticas em linguagem Python:

a. $2 + 3 \times 3$

b. $4^2 \div 3$

c. $(9^2 + 2) \times 6 - 1$

TEMA 3 – VARIÁVEIS, DADOS E SEUS TIPOS

Já sabemos que um computador contém uma memória RAM. Dessa forma, todo e qualquer programa computacional, quando em execução, recebe do sistema operacional um espaço na memória destinado à sua execução e armazena seus dados nesta região. Porém, o que são os dados?

Puga e Riseti (2016, p. 18) definem um dado como sendo uma sequência de símbolos quantificados ou quantificáveis. Dados são valores fornecidos pela entrada do programa que podem ser obtidos via usuário ou processamento e que são manipulados ao longo de toda a execução do algoritmo.

Dados devem ser armazenados em variáveis. Vamos imaginar a memória do computador como uma grande estante cheia de gavetas. Cada gaveta será representada por um nome de identificação, o qual chamamos de *variável*.

Uma variável é, portanto, um nome dado para uma região da memória do programa. Sempre que você invocar o nome de uma respectiva variável, seu bloco de memória será automaticamente carregado da RAM, e manipulado pela CPU.

Utilizamos variáveis em todos os nossos algoritmos, com o objetivo de armazenar os dados e processamento ao longo de nosso programa. As variáveis apresentam tipos, e cada tipo contém características distintas de operação e manipulação. Iremos investigar um pouco

mais esses tipos ao longo deste tema. De acordo com nossa bibliografia, citamos três tipos de

variáveis denominados **tipos primitivos de dados**:

- Numérico – é um tipo que serve para representar qualquer número, sejam valores inteiros, ou valores com casas decimais (também chamados de ponto flutuante). Uma variável deve ser declarada como numérica quando operações aritméticas precisarem ser realizadas com elas;
- Caractere – é um tipo que serve para representar letras, caracteres especiais, acentuações, e até mesmo podem ser utilizados para números, quando estes não servirem para operações aritméticas, por exemplo;
- Literal/booleano – é um tipo que serve para representar somente dois estados lógicos: verdadeiro ou falso (1 ou 0).

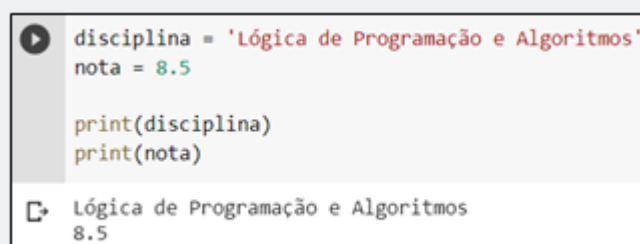
Saiba mais

Algumas bibliografias trabalham com quatro tipos primitivos, pois dividem o numérico em inteiro e real. O livro da Puga e Riseti (2016), *Lógica de programação e estrutura de dados*, é um exemplo de bibliografia que trabalha assim:

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.

Antes de entrarmos nos detalhes, vamos praticar um pouco. No código a seguir estamos criando duas variáveis (não vamos tratar do *tipo* ainda). A primeira tem nome *disciplina*, e armazena o nome da nossa disciplina. A outra se chama *nota* e armazena uma nota tirada nessa disciplina.

Figura 19 – Disciplina e nota



```
disciplina = 'Lógica de Programação e Algoritmos'
nota = 8.5

print(disciplina)
print(nota)
```


Lógica de Programação e Algoritmos
8.5

Observe que o sinal de igual é o que indica que o conteúdo colocado à direita será armazenado na variável colocada à esquerda. Em programação, o sinal de igual significa *atribuição*, ou seja, o valor é atribuído à variável.

Se pudéssemos tirar uma foto da memória do nosso programa, dentro do computador, veríamos esses dados colocados lá dentro, porém, representados de maneira binária.

Podemos apresentar na tela ambas variáveis em um mesmo *print*, e ainda intercalar com textos já prontos. Para isso, separamos cada parte por vírgula:

Figura 20 – Separando com vírgulas



```
print('Disciplina: ', disciplina, '. Nota:', nota)
```

Disciplina: Lógica de Programação e Algoritmos . Nota: 8.5

3.1 REGRAS PARA NOMES DE VARIÁVEIS

Em Python e na maioria das linguagens de programação, algumas regrinhas devem ser consideradas ao escolhermos os nomes para nossas variáveis, pois alguns símbolos e combinações não são aceitos.

Uma das regras mais comuns é a de que nomes de variáveis nunca podem iniciar com um número. Deve-se iniciar o nome de uma variável com uma letra ou o caractere especial de sublinha (_). Números, letras e sublinhas podem ser empregados à vontade no meio do nome. A partir da versão do Python 3.0, caracteres com acentuação são permitidos também. A Tabela 2 traz alguns exemplos de nomes de variáveis válidos e inválidos.

Tabela 2 – Exemplos de nomes de variáveis válidos e inválidos

Nome	Permitido?	Explicação
idade	Sim	Nome formado somente por letras/caracteres não especiais.
v3	Sim	Números são permitidos desde que não no início da palavra.

3v	Não	Não podemos iniciar uma variável com um número.
Maior_nota	Sim	Podemos usar caracteres maiúsculas e o sublinha sem problemas.
Maior nota	Não	O uso de espaços não é permitido em nomes de variáveis.
maior	Sim	Podemos usar a sublinha em qualquer parte da variável, inclusive no início.
#maior	Não	Caracteres especiais não são permitidos em nenhuma parte do nome da variável.
adicao	Sim	Nome formado somente por letras/caracteres não especiais.
Adição	Parcialmente	Somente o Python 3 permite caracteres com acentuação. Recomenda-se fortemente que evite esta prática, pois quase nenhuma linguagem admite isso.

3.2 VARIÁVEIS NUMÉRICAS

Uma variável é numérica quando armazena um número inteiro ou de ponto flutuante. Um número inteiro é aquele que pertence ao conjunto de números naturais e inteiros na matemática. Dentre os inteiros citamos: 100, -10, 0, 1569, -5629, 39999.

Já números de ponto flutuante armazenam casas decimais. O termo *ponto flutuante (float)* é oriundo da vírgula (na programação e na língua inglesa, usamos um ponto), que é capaz de deslocar-se pelo número (flutuando) e alterando o expoente do dado numérico. A quantidade de casas decimais armazenadas depende do tamanho da variável, enquanto mais decimais, mais memória é necessária. Dados de ponto flutuante podem ser: 100.02, -10.7, 0.00, 1569.999, -5629.312, 39999.12345.

Atenção! Um número de ponto flutuante que contenha casas decimais com valores todos zeros ainda continua sendo um valor de pontos flutuante, pois as casas decimais continuam existindo e ocupando memória. Por exemplo, 0.00 ou 123.0.

Saiba mais

Douglas Adams nos ensinou que a resposta para o sentido da vida, do universo e tudo mais é 42. Armazene em uma variável o sentido da vida. Crie uma mensagem que imprima na tela essa variável junto de uma frase dizendo que ela é o sentido da vida.

3.3 VARIÁVEIS LÓGICAS

Uma variável do tipo lógico, também conhecido como *booleana*, realiza operações lógicas empregando a álgebra proposta por George Boole. Uma variável lógica armazena somente dois estados, *verdadeiro* (*true*, em inglês) e *falso* (*false*, em inglês).

Saiba mais

George Boole foi um matemático britânico que viveu no século XIX. Criou a álgebra booleana, fundamental para a eletrônica digital e para o funcionamento e projeto de computadores.

Na memória do programa, ambos estados podem ser representados por um único *bit*. E esta é a grande vantagem deste tipo de variável, pois seu uso de memória no programa é ínfimo. O *bit* de valor 1 irá representar verdadeiro, e o *bit* de valor 0 representará falso.

Com uma variável booleana, podemos realizar operações lógicas. A Tabela 3 apresenta os operadores relacionais na linguagem Python, bem como o seu equivalente em pseudocódigo.

Tabela 3 – Lista de operadores lógicos em Python e em pseudocódigo

Python	Pseudocódigo	Operação
==	=	Igualdade
>	>	Maior que
<	<	Menor que
>=	>=	Maior ou igual que
<=	<=	Menor ou igual que
!=	<>	Diferente

A resposta para um cálculo lógico realizado com um dos operadores da Tabela 3 será sempre um valor booleano. Vejamos o exemplo em que a variável *a* contém o valor 1 e a variável *b* o valor

5.

Figura 21 – Valor booleano

```
▶ a = 1 #a recebe 1  
b = 5 #b recebe 5
```

Podemos realizar operações lógicas entre estas variáveis. A seguir, temos uma terceira variável, chamada de *resposta*, que recebe o resultado da comparação lógica de igualdade entre *a* e *b*. O que o computador faz é perguntar se *a* é igual a *b*. Como '*a* = 1' e '*b* = 5', não são iguais. Portanto, a resposta será falsa (*false*), e é apresentada como resultado e salva na variável de saída.

Figura 22 – Variável *resposta*

```
▶ #resposta recebe o resultado da comparação lógica de igualdade  
resposta = a == b  
print(resposta)  
False
```

Atenção! Veja que, quando utilizamos um único sinal de igualdade, isso significa atribuição. Já dois iguais é a comparação lógica de igualdade. Não confundam!

A seguir, temos a comparação lógica de diferenciação. Nesse caso, *a* é diferente de *b*? Sim, porque o valor 1 é diferente de 5, resultando em verdadeiro.

Figura 23 – Comparação lógica de diferenciação

```
▶ #resposta recebe o resultado da comparação lógica de diferente  
resposta = a != b  
print(resposta)  
True
```

Saiba mais

Crie uma variável que receba uma nota de um aluno. Crie outra variável que receba o resultado de uma comparação lógica entre a nota escolhida e o valor 7, que é a média para aprovação. Caso a nota seja maior ou igual a 7, o resultado deve ser verdadeiro. Imprima o resultado da comparação na tela.

3.4 VARIÁVEIS DE CADEIA DE CARACTERES (STRINGS)

É muito comum necessitarmos armazenar conjuntos de símbolos, como frases inteiras, em uma única variável. Ainda, muitas vezes precisamos envolver acentuação, pontuação, números e tabulação, tudo isso em uma só variável. Podemos representar e guardar tudo isso em uma variável denominada de *cadeia de caracteres* ou *string*. Antes de iniciarmos a manipular a variável do tipo *string*, vamos compreender um pouco como o computador sabe o significado de cada símbolo.

Cada símbolo que conhecemos é codificado e representado dentro do computador por uma sequência de *bits*, que segue um padrão internacional. O primeiro padrão foi criado na década de 1960, e é denominado *tabela ASCII (American Standard Code for Information Interchange)* (Ascii..., [S.d.]). Esse padrão define a codificação para 256 símbolos (8 *bits*), dentre os quais números, letras maiúsculas e minúsculas, mais alguns caracteres empregados em todas as línguas de maneira universal, como ponto de interrogação, assim como alguns símbolos de tabulação, como o recuo de parágrafo ou espaço simples.

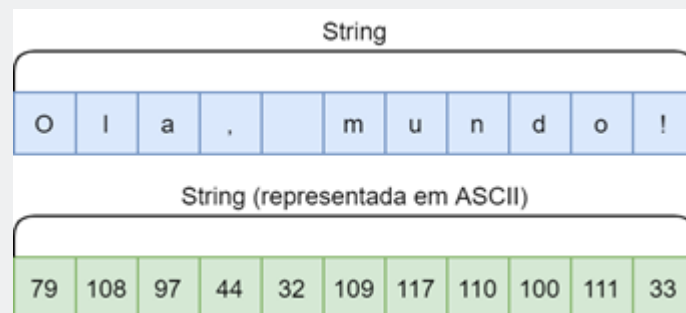
No entanto, rapidamente o padrão ASCII tornou-se obsoleto, uma vez que muitos caracteres ficaram de fora e não podiam ser representados, já que 8 *bits* não era suficiente. Assim se criou uma extensão a ele, chamada de *Unicode* ([S.d.]). Este padrão hoje é capaz de suportar mais de 4 bilhões de símbolos (32 *bits*). O Unicode é capaz de representar todos os caracteres orientais, acentuações de todas as línguas e até mesmo *emojis* têm codificação dentro desse padrão.

Vamos voltar às variáveis do tipo cadeia de caracteres. Veja, o Python armazena cada caractere em um espaço de memória próprio, que pode ser acessado individualmente (veremos isso em breve), porém, todo o espaço da variável é um bloco alocado sequencialmente na

memória destinada ao programa.

Se pudéssemos tirar uma foto interna da memória do nosso programa, veríamos uma sequência de dados como na Figura 24. Não obstante, eles estariam codificados em um padrão, como ASCII ou Unicode. Observe que até mesmo o símbolo de espaço apresenta uma codificação em ASCII. Veja também que a letra a está sem acento, pois o padrão ASCII não conhece acentuação.

Figura 24 – Representação de uma *string* com a frase *Olá, mundo!* e seu equivalente em ASCII



A linguagem Python manipula esta cadeia de uma maneira bastante simplificada para o desenvolvedor. Ademais, o Python traz recursos poderosos para manipulação e uso de *strings*, algo que grande parte das linguagens não é capaz de fazer nativamente. Vamos explorar mais esses recursos no Tema 4 e em aulas subsequentes. Podemos armazenar uma *string* em uma só variável no Python, veja:

Figura 25 – *String* em uma só variável no Python

```
frase = 'Olá, mundo!'
print(frase)
```

Olá, mundo!

Ocorre na maioria das linguagens de programação, e no Python não é diferente: podemos acessar partes específicas do texto da *string*. Isso é possível porque cada caractere é também tratado como um dado individual na memória. Podemos acessá-los através do que chamamos de índice da *string*.

O *índice* é um número que indica a posição do caractere na cadeia. O primeiro índice é sempre o valor zero. Portanto, se quisermos acessar o primeiro dado da *string* "frase", fazemos:

Figura 26 – Primeiro dado da *string* "frase"



Acessamos o índice zero chamando o nome da variável e colocando colchetes. Dentro dos colchetes inserimos o índice, que é um valor inteiro. O retorno disso faz com que somente a letra 'O', maiúscula, apareça.

E se quiséssemos o terceiro caractere? Bom, para acessarmos o terceiro caractere, indicamos o índice 2. Observe que o índice é sempre um número a menos do que a posição desejada, uma vez que nossos índices iniciam a contagem em zero:

Figura 27 – Índice 2



Na figura a seguir, temos um exemplo de *string* juntamente de seus respectivos índices de acesso. Observe que temos 11 caracteres nessa frase, portanto, os índices vão de 0 a 10, sempre um a menos que o tamanho.

Figura 28 – Representação de uma *string* com seus índices



TEMA 4 – MANIPULAÇÕES AVANÇADAS COM STRINGS

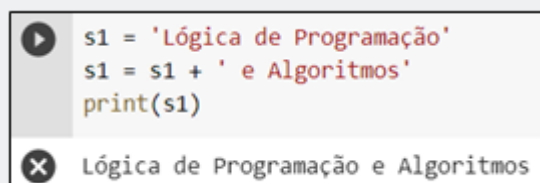
Você aprendeu quais são os tipos primitivos que o Python trabalha, porém *strings* apresentam um potencial bastante vasto a ser explorado na linguagem, pois podemos realizar operações como concatenação, composição e fatiamento de *strings*, tudo isso de maneira nativa com o Python. Vamos explorar um pouco mais o que significam estes recursos na linguagem de programação que estamos estudando e como podemos trabalhar com eles.

4.1 CONCATENAÇÃO

Concatenar *strings* significa juntá-las ou somá-las. Já fizemos concatenação anteriormente, no Tema 2, em que concatenamos frases dentro do *print*. Agora, vamos dar um passo além e trabalhar também com variáveis concatenadas.

A concatenação é realizada pelo operador de adição (+). Vejamos o exemplo a seguir. Criamos uma variável chamada de *s1*. Inicialmente, ela recebe parte de um nome. Em seguida, concatenamos o que já temos em *s1* com o restante, e imprimimos na tela:

Figura 29 - Concatenação

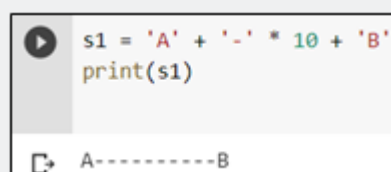


```
s1 = 'Lógica de Programação'
s1 = s1 + ' e Algoritmos'
print(s1)
```

Lógica de Programação e Algoritmos

Podemos repetir uma mesma *string* várias vezes na concatenação utilizando o símbolo de multiplicação (*). No exemplo a seguir, multiplicou-se o caractere do tracejado 10 vezes, facilitando sua escrita.

Figura 30 – Multiplicação



```
s1 = 'A' + '-' * 10 + 'B'
print(s1)
```

A-----B

Atenção! Não é possível realizar concatenação quando estivermos tentando juntar *strings* e variáveis numéricas. Ocorre erro de compilação. Para isso, utilize a composição.

Saiba mais

Imprima na tela uma variável do tipo *string* que escreva a seguinte frase abaixo. Crie a *string* concatenando tudo em uma só linha de código.

Linguagens de programação:

Python ---- C ---- Java ---- PHP

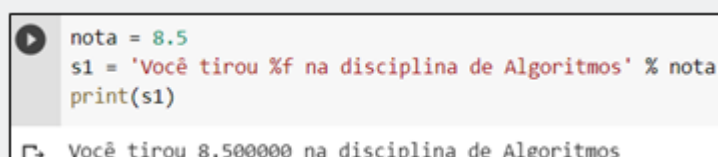
Para dar uma quebra de linha (*enter*), utilize `\n`. Para fazer uma tabulação (*tab*), utilize `\t`. Não esqueça de usar também o multiplicador de *strings*.

4.2 COMPOSIÇÃO

Já vimos anteriormente que podemos criar uma mensagem que mistura texto e dados numéricos em um *print*. Vamos agora ver mais detalhadamente o que existe no Python para nos auxiliar neste processo.

Podemos colocar o valor de uma variável dentro de uma outra variável que seja do tipo *string*. Para isso, iremos utilizar um símbolo de percentual (%), que vamos chamar de *marcador de posição*. Esse símbolo será colocado dentro de nosso texto, no local exato onde o valor de uma variável deve aparecer. Assim, ele irá marcar a posição da variável, que irá substituir o símbolo de percentual durante a execução do programa pelo valor da variável. Confuso? Vejamos um exemplo:

Figura 31 – Composição



```
nota = 8.5
s1 = 'Você tirou %f na disciplina de Algoritmos' % nota
print(s1)
```

Você tirou 8.500000 na disciplina de Algoritmos

Note que temos uma variável de *string* *s1*, a qual imprime na tela uma nota, portanto um dado numérico. Em um ponto específico da frase, existe o símbolo % seguido pela letra *f*.

Este percentual será substituído pelo valor da variável *nota*. A variável está posicionada após o término da frase, onde existe um símbolo de percentual e o respectivo nome. Isso significa que a variável *nota* terá seu valor inserido no lugar de %f dentro do texto.

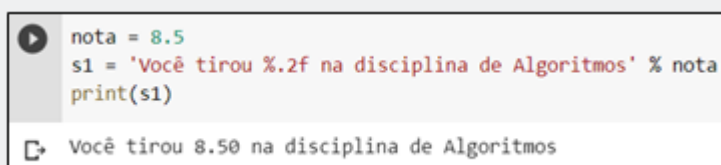
Note que a letra após o símbolo de percentual indica o tipo de dado que será colocado ali. Veja a Tabela 4 com as representações:

Tabela 4 – Lista de marcadores de posição

Marcadores	Tipo
%d ou %i	Números inteiros
%f	Números de ponto flutuante
%s	<i>Strings</i>

Se estivermos trabalhando com números de ponto flutuante, podemos delimitar quantas casas decimais queremos colocar na tela. Para isso, inserimos entre o símbolo de percentual e a letra do tipo da variável, um ponto e um número. O número indica quantas casas decimais teremos. Se usarmos .2, teremos duas casas decimais aparecendo.

Figura 32 – Casas decimais



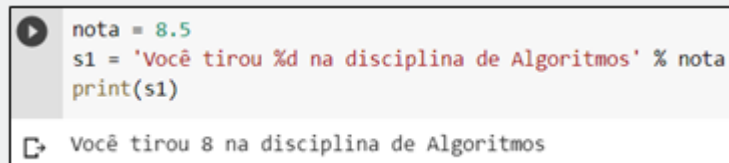
```
nota = 8.5
s1 = 'Você tirou %.2f na disciplina de Algoritmos' % nota
print(s1)
```

Você tirou 8.50 na disciplina de Algoritmos

Atenção para a escolha correta do tipo da variável. A variável *nota* contém casas decimais,

portanto é, obrigatoriamente, do tipo ponto flutuante e devemos usar `%f`. Se utilizarmos por engano o `%i` para números inteiros, o que acontecerá é que as casas decimais serão perdidas. Veja a seguir.

Figura 33 – Casas decimais perdidas

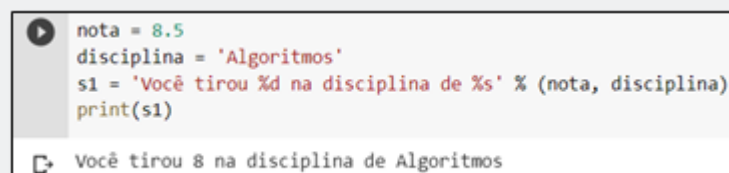


```
nota = 8.5
s1 = 'Você tirou %d na disciplina de Algoritmos' % nota
print(s1)
```

Você tirou 8 na disciplina de Algoritmos

Podemos colocar na frase quantas variáveis quisermos, basta colocarmos todos os símbolos de percentual em seus respectivos locais e apresentar o nome de todas as variáveis agora dentro de parênteses. Observe o exemplo a seguir com duas variáveis, uma do tipo ponto flutuante e outra do tipo *string*:

Figura 34 – Variável flutuante e variável *string*



```
nota = 8.5
disciplina = 'Algoritmos'
s1 = 'Você tirou %f na disciplina de %s' % (nota, disciplina)
print(s1)
```

Você tirou 8 na disciplina de Algoritmos

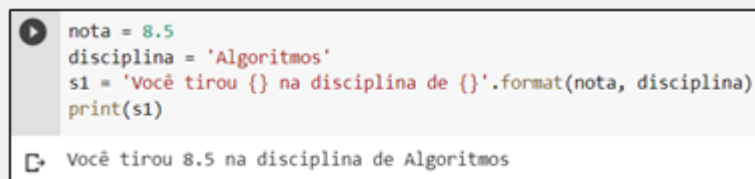
A maneira de composição apresentada até então, é uma herança da linguagem de programação C, linguagem que surgiu algumas décadas antes do Python e que influenciou a nossa linguagem em diversos aspectos.

A linguagem C é um pouco mais complexa de se trabalhar, pois é estritamente tipada e requer atenção maior aos detalhes. Deste modo, as versões mais recentes do Python passaram a adotar uma nomenclatura alternativa de composição, mais simples e menos dependente do desenvolver preocupar-se com o tipo da variável que está sendo impressa.

Em vez de `%` dentro do texto, usamos chaves. Ao invés do percentual fora do texto, usamos `.format`. Não trabalhamos com tipos aqui neste formato. E, caso necessário, as casas decimais

podem ser informadas dentro das chaves. Veja o exemplo a seguir.

Figura 35 – Nomenclatura alternativa de composição



```
nota = 8.5
disciplina = 'Algoritmos'
s1 = 'Você tirou {} na disciplina de {}'.format(nota, disciplina)
print(s1)
```

Você tirou 8.5 na disciplina de Algoritmos

Saiba mais

Crie três variáveis distintas. Uma contendo o nome da sua comida favorita. Outra contendo o seu ano de nascimento, e a terceira contendo o resultado da divisão do seu ano de nascimento pela sua idade.

Armazene em uma quarta variável do tipo *string* uma mensagem que contenha todas as informações das variáveis acima.

Resolva o exercício da maneira clássica de composição e também da maneira mais moderna.

4.3 FATIAMENTO

Já aprendemos como acessar um índice dentro da *string*. No entanto, podemos fazer mais do que isso. É possível manipularmos uma parte específica da cadeia de caracteres informando um intervalo de índices a ser lido, fatiando a *string*.

O intervalo é informado ao chamarmos o nome da *string* e indicando os índices de início e de fim, separadas por dois pontos (:). No exemplo a seguir, fatiamos a expressão *Lógica de Programação e Algoritmos*, mostrando na tela somente a palavra *Lógica*.

Figura 36 – Fatiamento 1



```
s1 = 'Lógica de Programação e Algoritmos'
```

Lógica

```
print(s1[0:6])
```

Lógica

Observe que a palavra *Lógica* está contida nos índices de zero até o cinco. Portanto, por que delimitamos o intervalo de zero até seis? O motivo para isso é que o índice final não é incluído no fatiamento, portanto ao colocarmos [0:6], estamos incluindo os índices de zero a cinco.

Vejamos outro exemplo. A palavra *Algoritmos* está colocada nos índices 24 até o 33 na *string* abaixo. No entanto, ao criarmos o fatiamento colocamos do 24 até o 34, caso contrário o último caractere não seria impresso.

Figura 37 – Fatiamento 2

```
s1 = 'Lógica de Programação e Algoritmos'
print(s1[24:34])
```

Algoritmos

É possível omitir o número da esquerda (início) ou da direita (final) para representar tudo a partir do início, ou tudo até o final, respectivamente. O primeiro exemplo, para imprimir somente a palavra *Lógica* poderia ser reescrito simplesmente como [:6], pois inclui tudo desde o início até o índice cinco. Do mesmo modo, para a palavra *Algoritmos*, poderíamos colocar somente [24:], pois inclui tudo a partir do índice 24 até o final.

Figura 38 – Fatiamento 3

```
s1 = 'Lógica de Programação e Algoritmos'
print(s1[:6])
```

Lógica

Finalizando, é também possível colocar somente [:], sem início, nem final. Dessa maneira, estaríamos escrevendo a *string* por completo na tela, somente. Também podemos escrever da

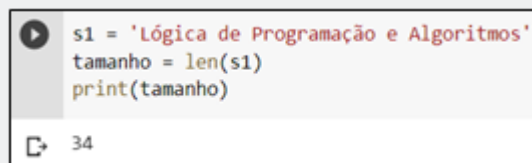
direita para a esquerda, basta usarmos valores negativos para os índices. O valor -1 seria o último caractere, -2 o penúltimo, e assim por diante.

4.4 TAMANHO (LEN)

Podemos descobrir quantos caracteres temos em uma *string* qualquer utilizando a função *len*, que é uma abreviação da *length*, que em inglês significa *tamanho*. Saber o tamanho de uma *string* pode ser bastante útil em diversas aplicações, especialmente quando precisarmos validar dados.

No exemplo a seguir, criamos uma variável denominada *tamanho* que recebe o resultado da função *len* de uma *string* *s1*. O resultado é que a *string* contém 34 caracteres. Atente-se ao fato de que caracteres como espaços também entram nesta contagem, afinal, também são codificados.

Figura 39 – Variável *tamanho*



```
s1 = 'Lógica de Programação e Algoritmos'
tamanho = len(s1)
print(tamanho)
```

34

Saiba mais

Crie uma variável de *string* que receba o seu nome completo. Crie uma segunda variável, agora do tipo booleana. Esta variável deverá receber o resultado da comparação lógica que verifica se o tamanho do seu nome é menor ou igual ao valor 15. Imprima a variável booleana na tela.

TEMA 5 – FUNÇÃO DE ENTRADA E FLUXO DE EXECUÇÃO DO PROGRAMA

Você aprendeu no Tema 2 como utilizar funções de saída. Em seguida, aprendeu a manipular

as variáveis e os dados em um programa. Agora, falta aprendermos a colocarmos dados de entrada via teclado em nosso programa, assim teremos um dinamismo maior em nossos exercícios.

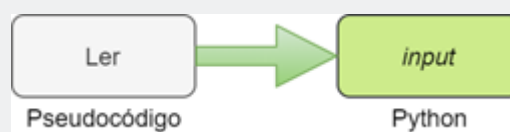
Neste último tema, vamos juntar os conhecimentos adquiridos para criarmos algoritmos um pouco mais elaborados e com mais aplicações práticas, portanto, prepare-se para colocar os dedos no teclado e programar.

5.1 FUNÇÃO DE ENTRADA

Todos os nossos exemplos até o momento sempre trabalharam com dados previamente conhecidos pelo nosso programa. Ou seja, todos os dados já estavam inseridos nas variáveis desde o início da execução do nosso algoritmo.

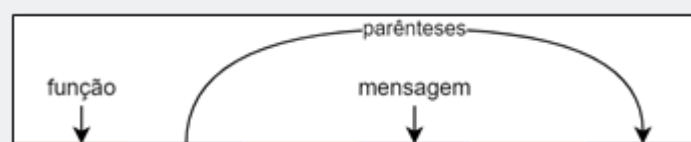
Vamos inserir mais um elemento em nossos algoritmos, o de permitir a inserção de dados por meio de um dispositivo de entrada, neste caso, o teclado. O comando de entrada de dados em pseudocódigo é chamado de *Ler*. é importante lembrar que o português estruturado é uma representação que independe de uma linguagem, e é empregado para estruturar ideias. Em linguagem Python, a função que provê esse recurso é chamada de *input*, conforme a Figura 40.

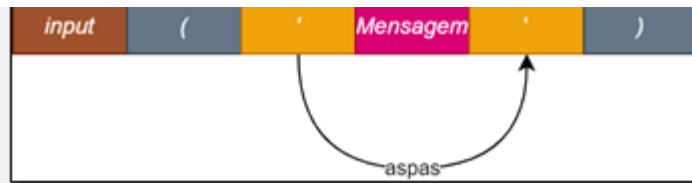
Figura 40 – Função de saída em pseudocódigo e em linguagem Python



Sua construção é muito semelhante à função de saída *print*. Utilizamos parênteses para delimitar uma mensagem para aparecerá na tela ao usuário, bem como usamos aspas simples para escrever a mensagem, conforme Figura 41.

Figura 41 – Detalhando a função *input* para leitura de dados via teclado





A função *input* irá, portanto, colocar na tela a mensagem de texto que você escolheu, e em seguida abrirá um campo para digitação. Nesse campo, o que for digitado será armazenado em uma variável definida na atribuição.

Vejamos um exemplo onde queremos perguntar ao usuário do programa qual é a sua idade. Para isso, colocamos no campo de mensagem a pergunta. Ainda, criamos uma variável que irá receber o que foi digitado, e assim armazenamos o dado para posterior uso.

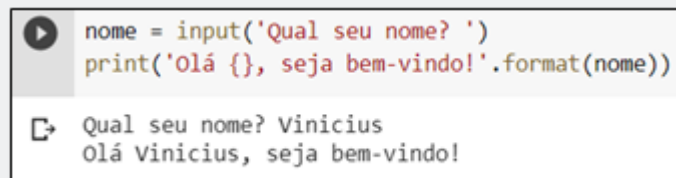
Figura 42 – Função *input*

Note que o cursor está esperando que digitemos a idade para avançar. Neste caso, o programa está travado na linha de código do *input*, e só irá avançar para a linha do *print* caso seja digitada alguma coisa e apertemos *enter*. Veja:

Figura 43 – Avançando para a linha *print*

Em um segundo exemplo, em que perguntamos um nome e, em seguida, usando composição de *strings*, escrevemos uma mensagem de boas-vindas ao usuário na tela. A mensagem será personalizada e mudará para cada nome digitado.

Figura 44 – Mensagem personalizada



```
nome = input('Qual seu nome? ')
print('Olá {}, seja bem-vindo!'.format(nome))
```

Qual seu nome? Vinicius
Olá Vinicius, seja bem-vindo!

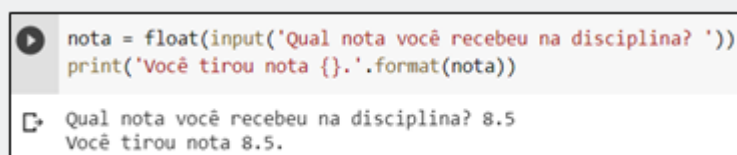
5.2 CONVERTENDO DADOS DE ENTRADA

Temos um pequeno problema que precisa ser resolvido. A função *input* sempre retorna para o programa um dado do tipo *string*. Isso significa que, caso precisemos de um dado numérico para ser utilizado posteriormente em algum cálculo, não iremos tê-lo, pois será uma *string*.

Para solucionar esse problema, precisamos converter o resultado de *input* em um valor inteiro ou ponto flutuante. Para isso, basta colocarmos a função *int* ou a função *float*, respectivamente, antes do *input*.

No exemplo a seguir, perguntamos a nota em determinada disciplina. Como a nota pode apresentar casas decimais, ela precisa ser ponto flutuante (*float*). Pronto, dessa maneira temos um dado numérico salvo em nosso programa e pronto para ser manipulado como tal.

Figura 45 – Perguntando a nota da disciplina



```
nota = float(input('Qual nota você recebeu na disciplina? '))
print('Você tirou nota {}'.format(nota))
```

Qual nota você recebeu na disciplina? 8.5
Você tirou nota 8.5.

5.3 FLUXO DE EXECUÇÃO DO PROGRAMA

Quando vimos a função de *input*, poucos parágrafos acima, você deve ter observado que, pela primeira vez, nosso programa ficou parado em uma linha específica do código, aguardando algo acontecer para prosseguir.

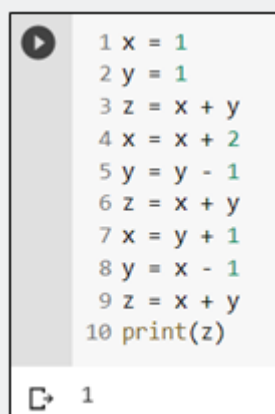
Como esta é a primeira vez que algo assim acontece, é interessante analisarmos um pouquinho como se dá a ordem de execução das coisas em um programa Python.

Nessa linguagem, cada linha indica uma instrução. Entenda por *instrução* qualquer comando que vimos até agora (entrada, saída e atribuição). Somente após uma linha ser executada por completo é que a próxima linha será executada.

O fluxo de execução em Python, assim como em todas as linguagens de programação tradicionais, se dá de cima para baixo. Ou seja, a instrução mais acima escrita será executada primeiro, ao finalizar, o programa desce para a linha de baixo.

Vamos compreender melhor o fluxo de execução do algoritmo por meio do exemplo abaixo. Não se assuste! Ele parece bastante complexo no início, mas tudo ficará claro. Foram inseridas numerações nas linhas para que possamos falar delas especificamente.

Figura 46 – Compreendendo o fluxo de execução do algoritmo



Nosso programa inicia sua execução de cima para baixo, ou seja, na linha 1, e vai descendo, uma a uma, até executar a linha 10, na qual imprime o valor de 'z' na tela. A cada linha executada, as variáveis 'x', 'y' e 'z' vão alterando seus valores neste exemplo.

Na Tabela 5, colocamos o que acontece com cada variável após executar cada linha. Note que somente uma variável sofreu mudanças em cada uma das linhas, pois é assim que o programa foi construído.

Tabela 5 – Valores das variáveis do programa linha após linha executada. NC = não criada ainda pelo programa. Em vermelho, está colocada a variável que sofreu alteração na respectiva linha

indicada

Linha	x	y	z
1	1	NC	NC
2	1	1	NC
3	1	1	2
4	3	1	2
5	3	0	2
6	3	0	3
7	1	0	3
8	1	0	3
9	1	0	1
10	1	0	1

Existem IDEs de compilação capazes de realizar o que chamamos de *depuração do código*, ou seja, a sua execução passo a passo, linha após linha, permitindo que o desenvolvedor enxergue as variáveis do seu programa durante todo esse processo. Nas aulas práticas, você verá como fazer isso em uma IDE de desenvolvimento.

Tenha em mente neste momento que compreender a ordem de execução do programa serve para que você consiga compreender como ele funciona, assim como auxilia você a desenvolver sua lógica e a encontrar possíveis problemas que estejam ocasionando mal funcionamento do seu algoritmo. Você pode, sempre que quiser, realizar um teste manual de execução do seu programa, chamado de *teste de mesa*, e acompanhar suas variáveis de maneira semelhante ao que foi feito na Tabela 5, mas sem a necessidade de um *software*.

5.4 RESOLVENDO EXERCÍCIOS

Vamos agora combinar o que aprendemos até então para a construção de algoritmos em alguns exercícios mais aplicados. Todos os exercícios aqui colocados estarão resolvidos e será

apresentado seu código em linguagem Python, assim como seu equivalente em pseudocódigo e seu respectivo fluxograma. A simbologia adotada para os fluxogramas está apresentada no livro da Puga e Riseti (2016, p. 14-15).

Na resolução dos exercícios em Python, em alguns momentos você encontrará textos antecidos por um símbolo de #. Sempre que isso aparecer, são comentários.

Comentários são textos que o desenvolvedor pode inserir em seu programa para auxiliar no seu entendimento. Seja porque você trabalha em equipe e precisa que seus colegas vejam o que você fez, ou somente para que você lembre depois do que havia feito. De todo o modo, comentários são completamente ignorados pelos compiladores e, portanto, podem ser utilizados à vontade, sem impacto algum no desempenho do seu programa.

Exercício 1

Desenvolva um algoritmo que solicite ao usuário dois números inteiros. Imprima a soma destes dois números na tela.

- Python

```
1 # Exercício 2.1
2 x = int(input('Digite um número inteiro: '))
3 y = int(input('Digite outro número inteiro: '))
4 # Maneira clássica
5 res = 'O resultado da soma de %i com %i é %i.' % (x, y, x + y)
6 print(res)
7 # Maneira Moderna
8 res = 'O resultado da soma de {} com {} é {}'.format(x, y, x + y)
9 print(res)
```

➤ Digite um número inteiro: 5
Digite outro número inteiro: 7
O resultado da soma de 5 com 7 é 12.
O resultado da soma de 5 com 7 é 12.

- Pseudocódigo

Algoritmo Exercício 2.1

Var

x, y, res: inteiro

Início

Escrever("Digite um número inteiro: ")

Ler(x)

Escrever("Digite outro número inteiro: ")

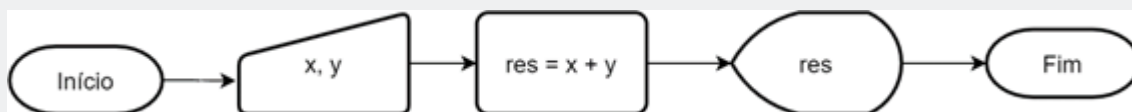
Ler(y)

res = x + y

Escrever("O resultado da soma de ", x, " com ", y, " é ", res)

Fim

- Fluxograma



Exercício 2

Desenvolva um algoritmo que solicite ao usuário uma quantidade de dias, de horas, de minutos e de segundos. Calcule o total de segundos resultante e imprima na tela para o usuário.

- Python

```
1 # Exercício 2.2
2 d = int(input('Quantos dias? '))
3 h = int(input('Quantas horas? '))
4 m = int(input('Quantos minutos? '))
5 s = int(input('Quantos segundos? '))
6 #1 dia = 24h | 1h = 60 min | 1 min = 60 s
7 total = s + (m * 60) + (h * 60 * 60) + (d * 24 * 60 * 60)
8 # Maneira clássica
9 res = 'O total de segundos calculado é de %i.' % total
10 print(res)
11 # Maneira moderna
12 res = 'O total de segundos calculado é de {}'.format(total)
13 print(res)
```

Quantos dias? 5
Quantas horas? 2
Quantos minutos? 30
Quantos segundos? 7
O total de segundos calculado é de 441007.
O total de segundos calculado é de 441007.

- Pseudocódigo

Algoritmo Exercício 2.2

Var

d, h, m, s, total: inteiro

Início

Escrever("Quantos dias? ")

Ler(d)

Escrever("Quantas horas? ")

Ler(h)

Escrever("Quantos minutos? ")

Ler(m)

Escrever("Quantos segundos? ")

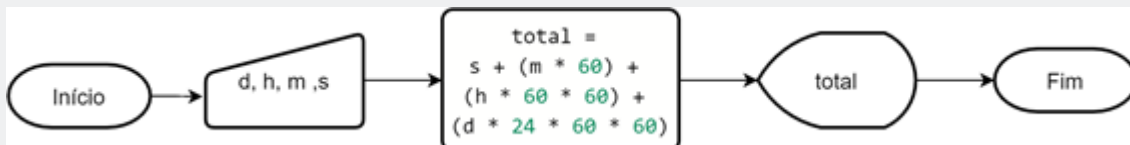
Ler(s)

$total = s + (m * 60) + (h * 60 * 60) + (d * 24 * 60 * 60)$

Escrever("O total de segundos calculado é de ", total)

Fim

- Fluxograma



Exercício 3

Desenvolva um algoritmo que solicite ao usuário o preço de um produto e um percentual de desconto a ser aplicado a ele. Calcule e exiba o valor do desconto e o preço final do produto.

- Python

```
1 # Exercício 2.3
2 preco = float(input('Digite o preço do produto: '))
3 p = float(input('Digite o percentual de desconto (0-100%): '))
4 desconto = preco * (p / 100)
5 final = preco - desconto
6 # Maneira clássica
7 print('O preço do produto é %.2f. Desconto de %.0f%%.' % (preco, p))
8 print('Valor calculado de desconto: %.2f. Valor final do produto: %.2f' % (desconto, final))
9 # Maneira moderna
10 print('O preço do produto é {}. Desconto de {}%.'.format(preco, p))
```

```
11 print('Valor calculado de desconto: {}. Valor final do produto: {}'.format(desconto, final))
```

Digite o preço do produto: 100
 Digite o percentual de desconto (0-100%): 25
 O preço do produto é 100.00. Desconto de 25%.
 Valor calculado de desconto: 25.00. Valor final do produto: 75.00
 O preço do produto é 100.0. Desconto de 25.0%.
 Valor calculado de desconto: 25.0. Valor final do produto: 75.0

- Pseudocódigo

Algoritmo Exercício 2.3 Var preço, p, desconto, final: real

Início

Escrever("Digite o preço do produto ")

Ler(preço)

Escrever("Digite o percentual de desconto (0-100%): ")

Ler(p)

desconto = preço * (p / 100)

final = preço - desconto

Escrever("O preço do produto é ", preço, ". Desconto de ", p, "%.")

Escrever("Valor calculado de desconto: ", desconto, "Valor final do produto:", final)

Fim

- Fluxograma



Exercício 4

Desenvolva um algoritmo que converta uma temperatura em Celsius (C) para Fahrenheit (F).

A equação de conversão é:

$$F = \frac{9 \times C}{5} + 32$$

- Python

```
1 C = float(input('Digite uma temperatura em Celsius: '))
2 F = (9 * C / 5) + 32
3 # Maneira clássica
4 print('Celsius: %.1f. Fahrenheit: %.1f' % (C, F))
5 # Maneira moderna
6 print('Celsius: {}. Fahrenheit: {}'.format(C, F))
```

☞ Digite uma temperatura em Celsius: 50
Celsius: 50.0. Fahrenheit: 122.0
Celsius: 50.0. Fahrenheit: 122.0

- Pseudocódigo

Algoritmo Exercício 2.4

Var

C, F: real

Início

Escrever("Digite uma temperatura em Celsius: ")

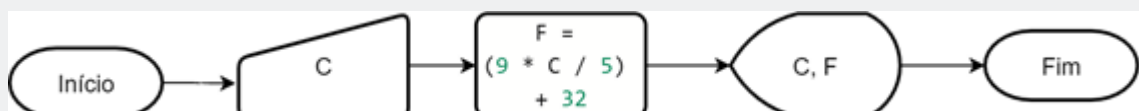
Ler(C)

$F = (9 * C / 5) + 32$

Escrever("Celsius: ", C, "Fahrenheit: ", F)

Fim

- Fluxograma



FINALIZANDO

Nesta aula, aprendemos a base que utilizaremos em qualquer programa em linguagem Python. Vimos o ciclo de processamento de um algoritmo e aprendemos a construir cada etapa dele.

Começamos de trás para frente, com a saída do programa e a função *print*. Depois

aprendemos a processar os dados, armazenando-os em variáveis e realizando operações aritméticas. Vimos também os tipos básicos de dados e trabalhamos com diferentes funções de manipulações de *strings*.

Para finalizar nossa aula, aprendemos a dar entrada de dados via teclado, com a função *input*. E resolvemos exercícios que envolvem todo o conhecimento adquirido nesta aula.

Reforçamos que tudo o que foi visto nesta aula continuará a ser bastante utilizado de maneira ao longo das próximas, portanto, pratique e resolva todos os exercícios do seu material.

REFERÊNCIAS

ASCII Table and Description. **Ascii Table**, [S.d.]. Disponível em: <<http://www.asciitable.com/>>. Acesso em: 6 set. 2020.

FORBELLONE, A. L. V. et al. **Lógica de programação: a construção de algoritmos e estruturas de dados**. 3. ed. São Paulo: Pearson, 2005

MATTHES, E. **Curso Intensivo de Python: uma introdução prática baseada em projetos à programação**. São Paulo: Novatec, 2015.

MENEZES, N. N. C. **Introdução à programação Python: algoritmos e lógica de programação para iniciantes**. 3. ed. São Paulo: Novatec, 2019.

PERKOVIC, L. **Introdução à computação usando Python** – Um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PUGA, S.; RISSETI, G. **Lógica de programação e estrutura de dados**. 3. ed. São Paulo: Pearson, 2016.

UNICODE. Disponível em: <<https://home.unicode.org/>>. Acesso em: 6 set. 2020.

