**Introdução - Linguagem de Programação**

Esta disciplina se propõe ao estudo do desenvolvimento de algoritmos e suas respectivas implementações na linguagem de programação Python, gerando programas executáveis em computadores. É uma das disciplinas fundamentais do curso e os conceitos estudados aqui servirão de base para todas as disciplinas relacionadas a programação. A disciplina está organizada em 4 unidades, nas quais serão abordados tópicos como: variáveis, tipos de dados, entrada e saída de dados, operadores aritméticos, relacionais e lógicos, estruturas de decisão, estruturas de repetição, funções e sequências.

As unidades são:

* Introdução, ambiente de desenvolvimento e estrutura básica de Python;
* Expressões relacionais e lógicas, estruturas de seleção e funções;
* Estruturas de repetição;
* Sequências (listas, tuplas e strings).

**Livros**

**Livro** - [Aprendendo com Python: Edição interativa](https://panda.ime.usp.br/pensepy/static/pensepy/#)

**Livro** - [Pense em Python (2ª ed.), de Allen B. Downey](https://penseallen.github.io/PensePython2e/)

**Parte 1 -** [**Introdução à disciplina e conceitos iniciais**](https://sites.google.com/faculdadeimpacta.com.br/lp-p1?authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=BTx-GIDSdvs)

**Resumo**

Nesta aula os objetivos são: (I) introduzir as diretrizes da disciplina, como conteúdo programático, metodologia de ensino, critérios de avaliação e bibliografia; (II) conceituar os termos “algoritmo”, “lógica de programação”, “linguagem de programação”, “código-fonte” e “programa”; (III) simular o processo para resolução de problemas computacionais sem o uso de computadores (computação desplugada).

**Motivação**

A vida das pessoas é constantemente alterada pela evolução das tecnologias, inclusive pela tecnologia computacional. Diversos aspectos da vida são impactados pelo progresso das descobertas relacionadas à área da Ciência da Computação. Você pode imaginar o quão diferente seria sua rotina sem a existência de notebooks e smartphones?

Brookshear (2008, p. 17) menciona que, como uma disciplina, “[a Ciência da Computação] busca construir uma base científica para diversos tópicos, tais como a construção e a programação de computadores, o processamento de informações, as soluções algorítmicas de problemas e o processo algorítmico propriamente dito.”. Podemos resolver problemas com a construção de algoritmos e, para automatizá-los, aplicaremos lógica de programação com instruções em linguagens de programação, criando códigos-fonte que se tornarão programas executados por computadores.

**Algoritmos**

Conforme descrito por Ziviani (1999, p. 1) “[algoritmo é] uma sequência de passos de ações executáveis para a obtenção de uma solução para um determinado tipo de problema”. Podemos imaginar um algoritmo como um procedimento que visa resolver um problema, formado por uma sequência ordenada e finita de passos passíveis de serem executados, assim como conceituado por Pereira (2010).

O termo algoritmo não está diretamente relacionado aos computadores, muito menos depende deles, pois podemos criar algoritmos que não serão, necessariamente, executados em um computador. Imagine, por exemplo, uma receita de bolo

Uma característica importantíssima sobre os algoritmos é que a construção pode variar de acordo com quem os elabora. Portanto, nem todos os algoritmos que objetivam resolver o mesmo problema terão a mesma sequência de passos, afinal dependerá, dentre outros fatores, da experiência, vocabulário, concisão e lógica do autor. Em contraste, a execução dos algoritmos, se forem bem formulados, independe de quem os executará, bastando que os passos sejam rigorosamente seguidos.

Genericamente, podemos dizer que lógica é uma parte da filosofia que trata das formas do pensamento em geral (dedução, indução, inferência, hipótese etc.) e das operações intelectuais que visam a determinação do que é verdadeiro ou falso. Faremos uso de lógica para escrever as instruções de nossos algoritmos, de modo que a execução de cada instrução faça sentido para atingir o objetivo final.

Segundo um dos mais importantes nomes da Computação, Knuth (1997), um algoritmo tem cinco características fundamentais:

**Finitude:** um algoritmo precisa terminar após um número finito de passos;

**Definição:** cada passo do algoritmo precisa ser precisamente definido, as ações devem ser especificadas clara e rigorosamente evitando ambiguidade;

**Entrada:** um algoritmo tem zero ou mais entradas, que são dados fornecidos ao algoritmo ao ser iniciado, ou dinamicamente conforme for executado;

**Saída:** um algoritmo tem zero ou mais saídas, que geralmente estão relacionadas com os dados fornecidos nas entradas;

**Eficácia:** é esperado que um algoritmo seja eficaz, no sentido que suas operações devem ser suficientemente básicas para que possam, a princípio, ser feitas precisamente e em um tempo finito por alguém com lápis e papel.

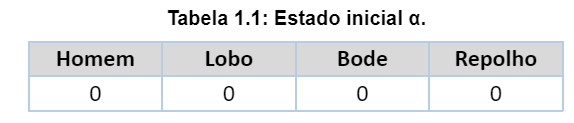
Nesta disciplina nossa atenção será direcionada para a lógica de programação, para que possamos estudar as estruturas básicas para a construção de algoritmos destinados a serem executados em computadores com uso de uma linguagem de programação.

**Abstração e representação**

Para resolver problemas computacionalmente são necessários pelo menos dois instrumentos: (I) uma representação que capte todos os aspectos relevantes do problema, descartando detalhes insignificantes para a sua resolução e; (II) um algoritmo que resolva o problema com base em ações aplicadas na representação construída.

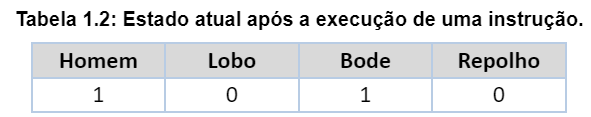
O que seria uma representação adequada para este problema? Uma vez que apenas os aspectos relevantes do problema precisam ser representados, todos os detalhes irrelevantes podem ser omitidos. Uma representação que deixa de fora detalhes do que está sendo representado é uma forma de abstração.

O uso de abstração é predominante na computação. Pense, a cor do barco é relevante? A largura do rio? O nome do homem? Não! A única informação relevante é onde está cada personagem/entidade desse contexto, a cada momento antes e após a execução de um passo/instrução da solução proposta para o problema. A localização coletiva de cada entidade, neste caso, refere-se ao estado do problema. Assim, o estado inicial do problema, que chamamos de α, pode ser representado conforme a Tabela 1.1.



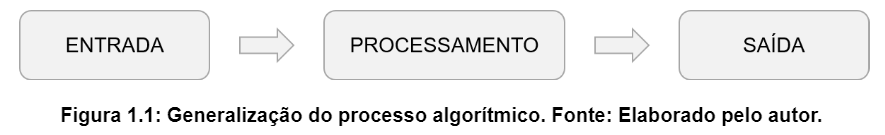
Note que nesta representação criamos uma tabela em que o nome das colunas são as entidades relevantes no contexto do problema, e os valores das colunas são indicações sobre os respectivos estados de cada entidade.

Arbitrariamente, definimos que colunas com 0 (zero) indicam que a respectiva entidade está no lado leste do rio. Já colunas com 1 (um) indicam que a entidade está no lado oeste do rio. Por exemplo, se a primeira instrução do algoritmo fosse “homem atravessa com o bode de leste para oeste”, a representação do novo estado do problema, gerado após a execução desta instrução, seria como ilustrado na Tabela 1.2. Neste novo estado apenas homem e bode estão no lado oeste e as localizações do lobo e do repolho não foram alteradas, pois a instrução executada não os impactou.



Com essa representação plenamente assimilada, podemos estabelecer que uma solução para este problema é a aplicação de um algoritmo que tenha como entrada o estado inicial α (Tabela 1.1), processe α com uma sequência de instruções logicamente ordenadas e que conduzam à saída do estado final β.

Portanto, podemos generalizar nossos algoritmos em três etapas: (I) receber entradas; (II) processá-las e; (III) gerar saídas que resolvam o problema de acordo com as entradas processadas.



É importante diferenciar a “representação” do “algoritmo” que será aplicado sobre ela. A tabela é apenas uma forma de representar o “mundo do problema” que queremos resolver, já o algoritmo é uma sequência de ações que serão aplicadas neste “mundo” de forma a deixá-lo no estado que desejamos, solucionando o problema.

**Linguagens de Programação**

Até agora discutimos sobre algoritmos de forma genérica, porém, o foco da disciplina é o estudo de algoritmos que serão executados em computadores, ou seja, serão transformados em programas. Thomas Cormen, conceitua esse tipo de algoritmo:

*Também podemos considerar um algoritmo como uma ferramenta para resolver um problema computacional bem especificado. O enunciado do problema específica em termos gerais a relação desejada entre entrada e saída. O algoritmo descreve um procedimento computacional específico para se conseguir essa relação entre entrada e saída. (CORMEN et al., 2012, p. 3).*

Para que os algoritmos possam ser executados automaticamente é necessário que sejam escritos em linguagens compreensíveis aos computadores, que são as linguagens de programação. A codificação, também chamada de implementação, é a fase em que o programador escreve seus algoritmos em uma linguagem de programação.

Como escrito por Lira (2019, p. 29), “Linguagens de programação são conjuntos de símbolos e regras de sintaxe que permitem a construção de instruções que descrevem, de forma não ambígua, ações que podem ser entendidas e executadas por meio de computadores”. Cada linguagem de programação define um padrão para a escrita de suas instruções, esses padrões têm motivações diversas, algumas linguagens favorecem a legibilidade para humanos, outras a concisão, ou o desempenho na execução etc.

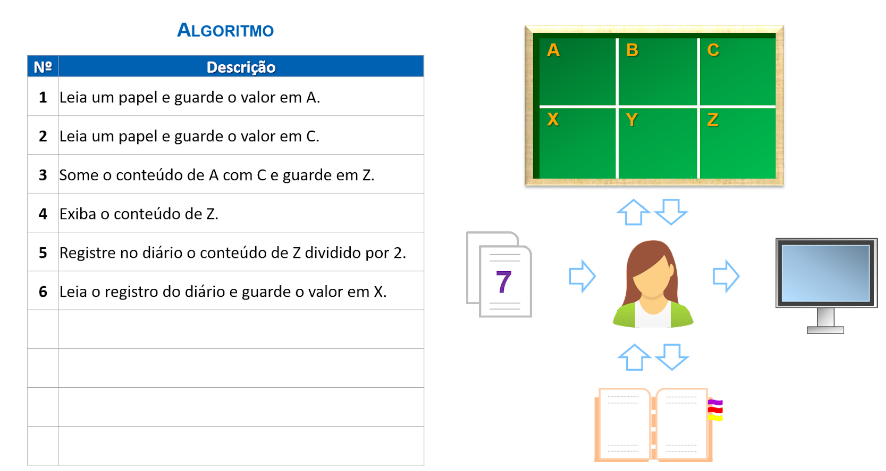
Em nossa disciplina utilizaremos a linguagem de programação Python 3.

**Computador Simplificado**

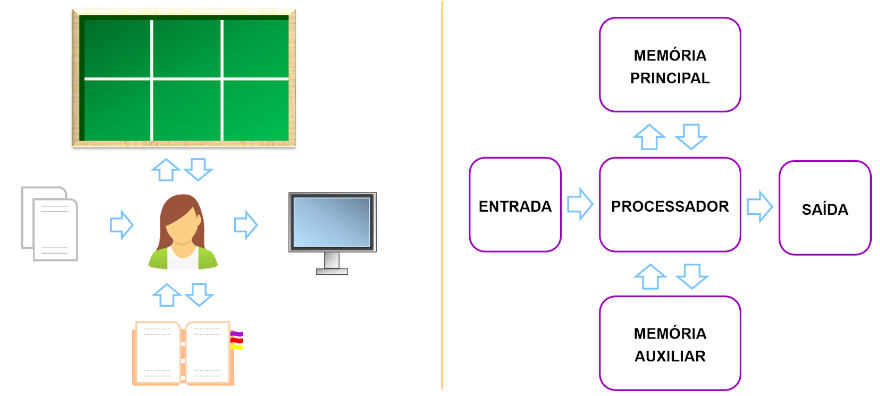
Podemos simular procedimentos computacionais sem a necessidade de um computador, aprendendo conceitos básicos e usando objetos do dia a dia que, em seguida, serão mapeados para correspondentes de um sistema computacional eletrônico. O Computador Simplificado (que abreviaremos como CS) se encaixa neste contexto!

À direita da Figura 1.2 está um algoritmo com seis instruções, à esquerda há uma representação simplificada de um sistema computacional sem referência direta aos componentes de um computador, apenas objetos do cotidiano com função equivalente.

No CS, temos: (I) ao centro, uma pessoa chamada Megan, responsável por executar as instruções do algoritmo; (II) à esquerda, papéis descartáveis, cada um representando uma entrada de dados; (III) à direita, uma tela para exibir dados para outras pessoas; (IV) acima, uma lousa em que cada parte tem um nome, em nosso caso uma única letra, e pode guardar dados temporariamente, afinal pode ser facilmente apagada; (V) abaixo, um diário para registrar dados, de modo a armazená-los por longos períodos. As setas ao redor de Megan indicam o fluxo em que os dados podem trafegar.



Com reconhecimento da importância de cada parte do CS, a Figura 1.7 ilustra a correspondência de seus itens em relação aos componentes de um computador real.



O mapeamento é dado assim: (a) Megan é o processador, pois é o componente que executa as instruções do algoritmo; (b) os papéis descartáveis são dados fornecidos como entrada, geralmente inseridos pelo usuário por meio de um teclado, mouse ou tela sensível ao toque; (c) a tela é a saída, que corresponde a um monitor ou impressora, por exemplo; (d) a lousa simboliza a memória principal, conhecida como Memória RAM, que serve para guardar dados temporariamente, pois ao cortar o fluxo de energia do computador os dados são apagados e; (e) o diário é a memória auxiliar, que serve para persistir os dados, ou seja, mantê-los registrados para que possam ser consultados futuramente, como HDDs, SSDs, cartões de memória, pendrives etc.

Você pode estar se perguntando: “qual seria a equivalência do algoritmo em um sistema computacional real?”. Resposta: ao código-fonte de um programa escrito em uma linguagem de programação. Aprender essa linguagem será um de nossos objetivos!

O que a maioria das escolas não ensinam. <https://youtu.be/nKIu9yen5nc>

**Parte 2 -** [**Ambiente de desenvolvimento, tipos de dados e variáveis**](https://sites.google.com/faculdadeimpacta.com.br/lp-p2?pli=1&authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=IVJ3ER-kmGg)

**Resumo**

Nesta aula os objetivos são: (I) preparar o ambiente computacional necessário para a disciplina; (II) acessar as duas ferramentas básicas utilizadas na disciplina para o desenvolvimento de programas em Python (interpretador interativo e editor de código fonte); (III) conceituar e utilizar tipos de dados primitivos, constantes e variáveis; (IV) conhecer os operadores aritméticos; (V) manipular o sistema online Python Tutor.

**Motivação**

Para criarmos nossos programas, teremos que usar um ambiente que permita escrever algoritmos em uma linguagem de programação e que possibilite a execução automática dessas instruções. Por isso, nesta aula faremos o download, instalação e execução do interpretador da linguagem Python e do IDLE, um software que possibilita o processo básico para criação e execução de programas em Python. Introduziremos conceitos básicos de Python para construir instruções simples e, por fim, manipularemos um sistema online que possibilita a visualização passo a passo do que ocorre quando nossos códigos-fonte são executados, potencialmente facilitando a aprendizagem.

**Download e instalação do Python no Windows**

Iniciaremos com o download do interpretador do Python juntamente com o IDLE, ambos são softwares gratuitos e de código aberto e serão instalados por meio do mesmo arquivo. Para isso basta acessar o site https://www.python.org, clicar na guia “Downloads” e, em seguida, no botão “Python 3.9.1” (atualmente a última versão estável disponível), como ilustrado na Figura 2.1.

**IDLE**

O IDLE (Ambiente Integrado de Desenvolvimento e Aprendizagem, tradução nossa) é o ambiente em que programaremos e foi instalado juntamente com o interpretador do Python no procedimento anterior. Este ambiente é programado na própria linguagem Python e funciona igualmente em outros sistemas operacionais. O IDLE possui dois módulos, o interpretador interativo e o editor de código-fonte, analisaremos as funcionalidades de cada um deles.

**Interpretador interativo**

Por padrão, ao clicar no atalho do IDLE no Menu Iniciar a tela exibida será do interpretador interativo, também conhecido como Shell.

Este módulo é usado principalmente para testar instruções de Python e executar programas construídos no editor. Repare na sequência de três sinais de maior (>>>) indicando que a Shell está pronta para receber novas instruções. A cada instrução, o programador deve teclar [ENTER] para executá-la e gerar o efeito correspondente.

A Shell é bastante útil e possui atalhos e recursos interessantes, podendo, inclusive, substituir uma calculadora tradicional, veremos posteriormente como fazer isso. Alguns dos atalhos que facilitam a codificação são:

* Para regressar a última instruções inserida, posicione o cursor na linha com >>> e pressione as teclas [ALT]+[P];
* Para avançar para a próxima instrução inserida, posicione o cursor na linha com >>> e pressione as teclas [ALT]+[N];
* Para interromper a execução de uma instrução ou programa pressione as teclas [CTRL]+[C].

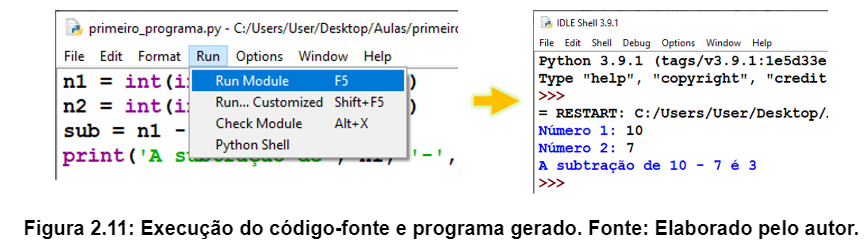
**Editor de código-fonte**

A Shell é muito útil para testar trechos de código, mas existem situações em que precisaremos construir um programa completo e guardar seu código-fonte para que possa ser editado e executado diversas vezes sem a necessidade de reescrevê-lo após encerrar o IDLE. Para isso existe o módulo editor de código-fonte. Para acessá-lo, clique no menu “File > New File” da janela da Shell.

Cada caractere é importante, portanto tome cuidado para não esquecer de nenhum ou trocá-lo por acidente. Caso tenha dúvida se em Python letras maiúsculas e minúsculas são interpretadas como caracteres diferentes, a resposta é SIM! Python é uma linguagem case-sensitive, isso quer dizer que n1 é completamente diferente de N1.

Repare que a barra de título da janela consta como “untitled”, indicando que o arquivo não foi salvo. Para executarmos o código-fonte e visualizar o programa funcionando, precisamos salvar este arquivo. Para isso, acesse o menu “File > Save”, defina o nome do arquivo e o local em que será salvo.

Após o salvamento do arquivo, a barra de título será alterada para corresponder ao nome do arquivo e local em que foi salvo. Para executar o código-fonte e visualizar o programa funcionando, vá até o menu “Run > Run Module”. O procedimento de execução e o resultado, supondo as entradas 10 e 7, estão ilustrados na Figura 2.11. Note que as saídas do programa estão em azul e as entradas do usuário em preto.



Observe que a execução do programa, mesmo que solicitada no editor, é feita na Shell. Caso queira fazer alguma alteração no código, basta selecionar novamente a janela do editor, que provavelmente foi sobreposta pela da Shell, fazer as modificações e executá-lo novamente, o código-fonte será automaticamente salvo.

**Conceitos básicos de Python**

Agora que temos conhecimento sobre o ambiente de programação que usaremos durante a disciplina, vamos aprender alguns conceitos básicos da linguagem Python.

**Constantes ou Literais**

Constantes são símbolos que representam valores e não podem ser alterados, são também chamadas de literais e geralmente são utilizadas em expressões. Exemplos de constantes em Python: 9, -541, 3.1415, 'estou aprendendo Python!', "Olá Megan", False e True.

O Python possui alguns tipos de constantes, e cada tipo deve ser usado de acordo com o contexto do problema. Os quatro tipos que usaremos de início são:

* **Números inteiros (int):** valores numéricos que não possuem ponto decimal.

Exemplos: 13, 123456789, -65, 0, 0b1011(base 2), 0o7(base 8), 0xF(base 16).

* **Números reais (float):** valores numéricos com ponto decimal ou escritos em notação científica. Note que é utilizado um ponto, não uma vírgula.

Exemplos: 2.7345, .25, -65.0, 0.0, 6.02e-23, 6.02E-23, 2e1.

* **Valores booleanos (bool):** também conhecidos como valores lógicos, existem apenas dois valores, um corresponde a falso e o outro a verdadeiro.

Exemplos: False e True. Note que somente a primeira letra é maiúscula e não há aspas como nas strings a seguir.

* **Textos (string):** são cadeias de caracteres delimitadas por apóstrofos ou aspas. São usadas para representar textos (letras, palavras ou frases).

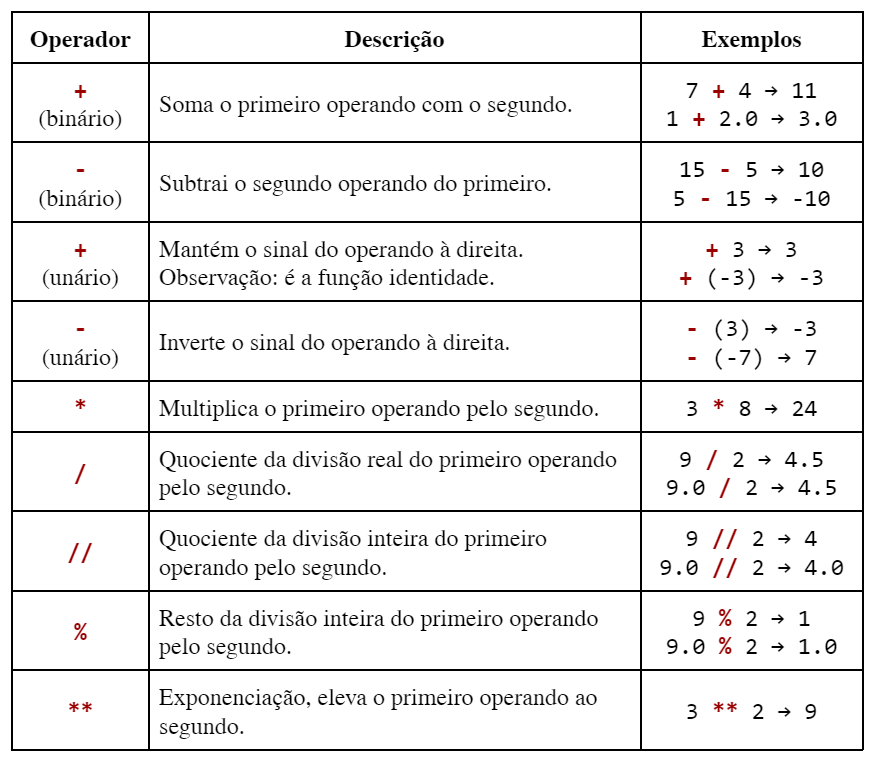
Exemplos: 'Olá!', '123', "Python é 10", "True", "3+5", 'X'.

**Operadores**

Em Python, assim como em outras linguagens de programação, operadores são símbolos pré-definidos que realizam uma operação sobre um ou mais operandos, produzindo um valor como resultado. O tipo do valor resultante dependerá do operador e dos operandos envolvidos. Quando um operador realiza uma operação entre 2 operandos, chamamos-o de binários, quando realiza uma operação com apenas 1 operando, chamamos-o de unários e, por fim, ternários quando envolvem 3 operandos.

**Operadores aritméticos**

Os operadores aritméticos, quando aplicados em operandos numéricos resultam valores numéricos, assim como na matemática.

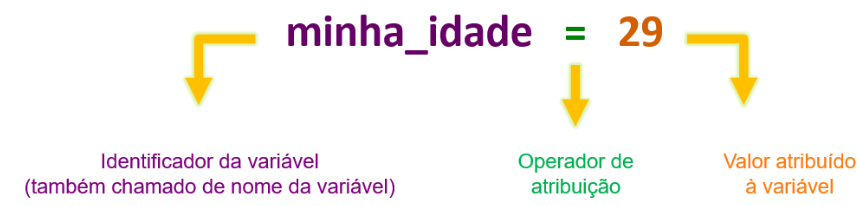


Note que, exceto pela divisão real, quando todos os operandos envolvidos na operação forem números inteiros o resultado será um número inteiro, porém, basta um operando real para que o resultado da operação resulte em um número real.

**Variáveis**

Uma variável é um espaço de memória associado a um identificador, ou seja, um nome, e serve para guardar valores que o programa poderá acessar e modificar. Toda variável possui um identificador único, de forma que possa ser referenciada pelo programador sem ambiguidade em qualquer parte do programa.

Em Python, uma variável é criada no momento em que um valor é atribuído a um identificador válido. A atribuição é feita colocando um identificador à esquerda de um sinal de igual e um valor à direita deste mesmo operador.



O conteúdo de uma variável pode “variar”, ou seja, uma mesma variável pode guardar valores diferentes em momentos diferentes de um programa. Lembre-se: uma variável só guarda um valor por vez, portanto a cada nova atribuição o valor atual será sobrescrito pelo novo.

Um destaque importante da linguagem Python é que o tipo do dado está relacionado ao valor atribuído e não a variável que recebeu esse valor, diferentemente de outras linguagens de programação como C, C++ e Java, dentre outras. Note também que uma mesma variável pode armazenar, inclusive, valores de tipos distintos.

**Identificadores**

Um identificador de uma variável, também referido como nome, é formado por uma sequência de um ou mais caracteres, de acordo com as seguintes regras:

Simplificadamente, pode conter apenas combinações de letras, dígitos e sublinhados (não pode conter símbolos especiais como &, ¨, %, $, #, @, !);

* Não pode iniciar com dígito;
* Não pode ser uma palavra reservada (abordaremos a seguir).

Recomenda-se criar identificadores concisos, porém descritivos:

* idade é melhor que i;
* tamanho\_nome é melhor que tamanho\_do\_nome\_da\_pessoa.

Evite abreviar exageradamente, escreva por extenso para melhorar a legibilidade:

* sobrenome é melhor que sbrnome;
* litros é melhor que ltrs;
* data\_criacao é melhor que dt\_cri.

Existe um guia de estilo para programação em Python, criado e mantido pela própria comunidade mundial da linguagem, com o intuito de fornecer recomendações para facilitar a leitura do código. Esse guia é conhecido como [PEP8](https://peps.python.org/pep-0008/) e embasou a forma como foram escritos os códigos deste material.

No decorrer da disciplina, apresentaremos as partes relevantes do guia para que a sua utilização ocorra de maneira natural, não sendo necessário decorá-lo integralmente. Por enquanto, temos mais uma recomendação relativa à criação de identificadores:

* Use apenas letras minúsculas e sem acentuação, separando as palavras com um sublinhado para melhorar a legibilidade.

Vale relembrar que o Python é uma linguagem case-sensitive, diferenciando letras maiúsculas de minúsculas, portanto o identificador meu\_nome não é o mesmo que Meu\_Nome ou MEU\_NOME.

**Palavras reservadas**

Python possui palavras reservadas, as palavras-chave, chamadas em inglês de keywords, e não podem ser usadas como identificadores, pois têm papel especial na linguagem. O Python 3.9.1 possui 36 keywords, porém a quantidade varia entre as versões, para saber quais são as de sua versão execute na Shell a Codificação 2.3.

>>> help('keywords')

Comentários

Podemos inserir comentários em nossos códigos-fonte, algo útil para ajudar tanto outros programadores que lerão nossos códigos quanto a nós mesmos para recordar a razão de determinadas instruções. Comentários são ignorados na execução do programa, portanto é algo para auxiliar humanos, não computadores. Linhas iniciadas com # são interpretadas como comentários.

**Parte 3 -** [**Funções embutidas, entrada e saída, operadores e expressões aritméticas**](https://sites.google.com/faculdadeimpacta.com.br/lp-p3?pli=1&authuser=2) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=GdBOkB5hhc8)

**Resumo**

Nesta aula os objetivos são: (I) relembrar expressões; (II) conhecer os operadores de atribuição composta; (III) compreender o conceito de precedência e associatividade de operadores; (IV) entender como Python avalia expressões; (V) descobrir o que são funções e como usar funções integradas; (VI) exibir dados para o usuário; (VII) obter entradas do usuário; (VIII) conhecer tipos de erros comuns em programação.

**Motivação**

Como já aprendemos o que são expressões, relembraremos pontos importantes e conheceremos outros, principalmente para compreender a forma que Python avalia expressões. Poderemos até substituir nossas calculadoras pela Shell, que possui muitas facilidades! Abordaremos funções, um recurso de programação que possibilita, por exemplo, usar códigos de outros programadores em nossos programas. Conheceremos funções para entrada e saída de dados, permitindo programas mais personalizados. Por fim, conheceremos alguns tipos de erros comuns na criação de programas.

Expressões

Uma expressão é uma combinação de operandos com zero ou mais operadores, resultando em um valor, sendo que os operandos podem ser, por exemplo, constantes e variáveis. Para que uma expressão esteja correta, essa combinação precisa ser válida. Um único operando já é considerado uma expressão válida. Também podemos usar parênteses, como na matemática.

>>> 3.1415 # expressão válida, resulta em 3.1415

>>> (2 + 5) # expressão válida, resulta em 7

>>> 3.0 \* 2 # expressão válida, resulta em 6.0

>>> 10 / 0 # expressão válida, porém resulta em erro de aritmética (erro de divisão por zero)

>>> +123 # expressão válida, resulta em 123

>>> 'oi' # expressão válida, resulta em 'oi'

>>> 123+ # expressão inválida, falta um operando à direita

>>> 2 \* 3) # expressão inválida, parênteses desbalanceados

Quando uma instrução contém uma expressão, Python avaliará a expressão, ou seja, encontrará o valor resultante, caso seja válida. Em expressões inválidas, ocorrerão erros que podem variar de acordo com a razão de estarem inválidas.

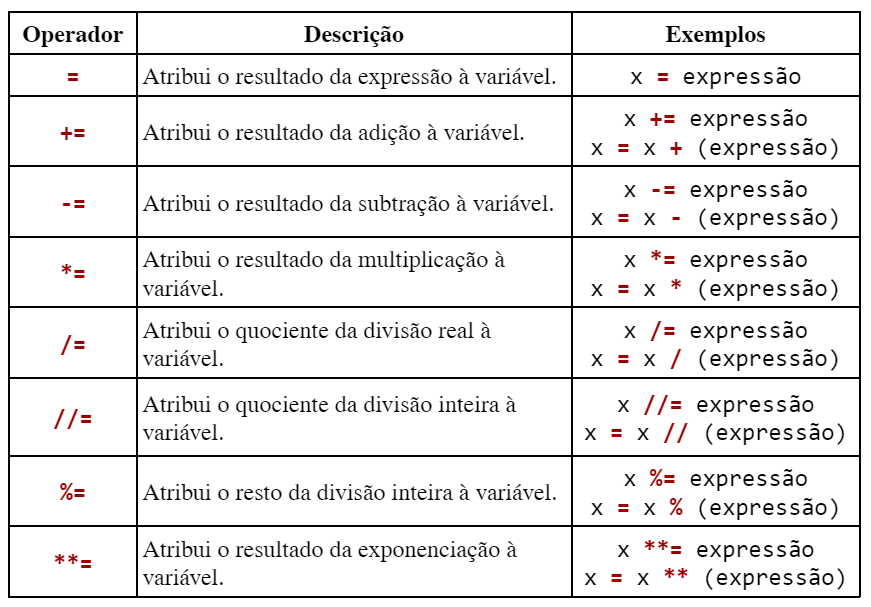
Conhecemos o operador de atribuição simples de Python (=), e entendemos que seu papel é atribuir valores de expressões às variáveis.

>>> x = 10 + 5 # atribui o valor 15 à variável x

É importante compreender que antes da atribuição, a expressão à direita de = será avaliada. Logo, a variável x não receberá a expressão 10 + 5, mas sim o valor resultante da avaliação de 10 + 5, que é 15. O mesmo ocorre em qualquer atribuição.

Em Python, há outros operadores de atribuição, chamados de operadores de atribuição composta, ou atribuição aumentada, usados para tornar algumas instruções mais concisas e, dependendo da maturidade do programador, mais legíveis. Também é possível substituir instruções com atribuição composta por atribuição simples combinada com outro operador.

Podemos dizer que estes operadores são um atalho de sintaxe para quando precisamos acessar o valor de uma variável, realizar uma operação com ele, e atribuir o resultado da operação à mesma variável.



>>> m = 2 # m recebe 2

>>> m \*\*= 5 # m recebe 32

>>> m //= 4 # m recebe 8

>>> m += 1 # m recebe 9

>>> m /= 2 # m recebe 4.5

>>> m %= 2 # m recebe 0.5

>>> m -= 0.2 # m recebe 0.3

>>> m \*= 5 # m recebe 1.5

###### **Codificação 3.3: Uma mesma variável recebendo diversas atribuições.**

>>> a = 10 # a recebe 10

>>> b = 10 # b recebe 10

>>> c = 10 # c recebe 10

>>> a \*= 2 + 3 # a recebe 50

>>> b = b \* 2 + 3 # b recebe 23

>>> c = c \* (2 + 3) # c recebe 50

###### **Codificação 3.4: Atribuição composta e atribuição simples equivalente.**

Talvez você tenha achado estranho que o segundo valor atribuído à variável a seja diferente do segundo valor atribuído à variável b, porém note que na terceira coluna da Tabela 3.1 está definido que a equivalência dos operadores de atribuição composta com a atribuição simples (=) só é atingida ao colocarmos a expressão, originalmente à direita, entre parênteses. Logo, a segunda atribuição à c corresponde à segunda de a.

É importante conhecer esse conjunto de operadores de atribuição, pois são frequentemente usados no mercado de trabalho, em fóruns e livros. Ao longo desta disciplina, usaremos principalmente os operadores =, += e -=.

Novamente, teste a Codificação 3.5 na Shell e verifique se compreendeu a razão dos valores atribuídos às variáveis.

>>> x = 10 # x recebe 10

>>> y = 10 # y recebe 10

>>> z = 10 # z recebe 10

>>> x += 2 \* 3 # x recebe 16

>>> y = y + 2 \* 3 # y recebe 16

>>> z = z + (2 \* 3) # z recebe 16

###### **Codificação 3.5: Atribuição composta e atribuição simples equivalente.**

Você pode ter ficado com dúvida sobre a razão das três últimas atribuições terem o mesmo resultado, mesmo não existindo parênteses na segunda atribuição para y. Isso ocorreu porque a precedência da multiplicação é naturalmente superior à da adição, como na matemática, tornando a ausência de parênteses irrelevante nesta instrução. Para entender plenamente como Python avalia expressões, é necessário estudarmos como essa linguagem de programação define a precedência e associatividade dos operadores.

**Precedência e associatividade de operadores**

Em Python, de modo semelhante à matemática, cada operador possui uma precedência, que é uma indicação de sua prioridade em relação aos outros operadores que estejam na mesma expressão.

Portanto, partes de uma expressão com operadores de maior precedência são avaliadas antes daquelas com operadores de menor precedência. Porém, a precedência natural de um operador pode ser artificialmente modificada com o uso de pares de parênteses, da mesma forma como fazemos na matemática.

Por exemplo, a expressão 4+1\*5 resulta em 9, afinal a multiplicação tem precedência sobre a adição e por isso será avaliada primeiro, mesmo que a adição esteja escrita antes na expressão. Lembre-se que a leitura de qualquer instrução é feita da esquerda para a direita, por isso apontamos que “a adição apareceu antes na expressão”.

Neste segundo exemplo, a expressão (4+1)\*5 resulta em 25, pois a precedência do operador de adição que está entre os parênteses foi alterada, aumentando sua prioridade. Em expressões com pares de parênteses aninhados, como (4+(1\*5))/2, a precedência é do mais aninhado para o menos aninhado. Uma conclusão natural, afinal o par de parênteses mais externo aumentou a prioridade da parte da expressão que está dentro dele e que, por sua vez, contém outro par de parênteses, aumentando a prioridade da parte da expressão dentro deste segundo par, e assim sucessivamente.

Porém, existem casos em que uma expressão pode conter operadores de mesma precedência, por exemplo, 8/4\*2. Repare que nesta expressão não podemos nos basear na precedência para decidir qual parte da expressão Python avaliará primeiro, afinal divisão e multiplicação têm a mesma precedência. No entanto, se a expressão contivesse parênteses, não haveria dúvidas, pois (8/4)\*2 resultaria 4.0, e 8/(4\*2) resultaria 1.0. Para resolver o impasse existe o conceito de associatividade.

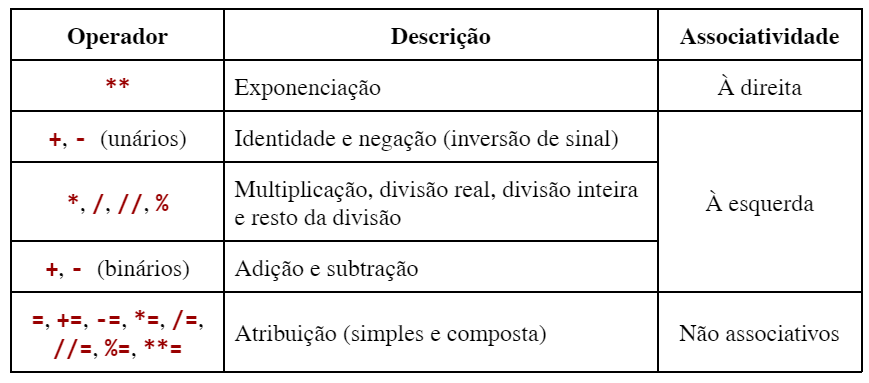
A associatividade é uma propriedade que define como operadores de mesma precedência devem ser agrupados, caso estejam na mesma expressão e sem parênteses que determinem a ordem da avaliação.

Por exemplo, na expressão 8/4\*2 o operando 4 está precedido e sucedido por operadores de mesma precedência, portanto poderia ser associado ao operador da esquerda, impondo que a avaliação começaria por 8/4, ou poderia ser associado ao operador da direita, impondo que a avaliação começaria por 4\*2. Com fundamento na regra de associatividade dos operadores de divisão real e multiplicação, poderemos concluir, sem ambiguidade, o resultado da expressão. Neste exemplo, o resultado será 4.0, porque esses dois operadores são associativos à esquerda.

Os operadores podem ser associativos à esquerda, associativos à direita ou não associativos. Veja a descrição de cada tipo:

* **Associativos à esquerda:** as operações são agrupadas da esquerda para a direita. Portanto, o operando será associado ao operador à sua esquerda;
* **Associativos à direita:** as operações são agrupadas da direita para a esquerda. Portanto, o operando será associado ao operador à sua direita;
* **Não associativos:** as operações não podem ser encadeadas, por uma de duas razões: (I) porque possuem comportamento indefinido, gerando um erro ou; (II) porque representam algo “especial”, uma interpretação diferenciada na linguagem.

Cada linguagem de programação possui suas próprias regras de precedência e associatividade de operadores, por isso é recomendado consultar o manual da linguagem para assegurar que as expressões coincidam com o desejado. A Tabela 3.2 possui uma relação simplificada dessas regras em relação aos operadores vistos até agora. Operadores na mesma linha possuem a mesma precedência e associatividade. A ordem de precedência está decrescente, logo o primeiro operador tem maior precedência e o último tem menor precedência.



**Avaliação de expressões**

De posse do conhecimento sobre precedência e associatividade de operadores e dos demais tópicos estudados até o momento, temos condições de compreender o procedimento de avaliação de expressões executado pelo Python. Para isso, analisaremos algumas expressões, avaliando-as de modo semelhante ao feito em Python.

>>> a = 5

>>> b = 4

>>> c = 9

>>> d = 7

>>> e = 1

>>> f = 2

>>> s = 10

>>> s += a + b \*\* (c - d) / e \* f

>>> s # qual o valor de s?

O par de parênteses prioriza a expressão interna, dando a ela maior precedência, porém, como a operação contém variáveis, primeiro devemos substituí-las por seus respectivos valores. Após ser avaliado, o par de parênteses é descartado:

s += ⇨ s += a + b \*\* 2 / e \* f

Precedência: s += a + 4 \*\*a + b \*\* (9 - 7) / e \* f 2 / e \* f ⇨ s += a + 16 / e \* f

Precedência e associativ.: s += a + 16 / 1 \* f ⇨ s += a + 16.0 \* f

Precedência: s += a + 16.0 \* 2 ⇨ s += a + 32.0

Precedência: s += 5 + 32.0 ⇨ s += 37.0

Único operador: s += 37.0

⇨ Atribuição composta, pode ser reescrita como: s = s + (37.0)

⇨ s = 10 + (37.0) ⇨ s = 10 + 37.0 ⇨ s = 47.0

>>> t = 1 + 2 - 3 + 4 - 5

>>> t # qual o valor de t?

Associatividade: t = 1 + 2 - 3 + 4 - 5 ⇨ t = 3 - 3 + 4 - 5

Associatividade: t = 3 - 3 + 4 - 5 ⇨ t = 0 + 4 - 5

Associatividade: t = 0 + 4 - 5 ⇨ t = 4 - 5

Associatividade: t = 4 - 5 ⇨ t = -1

Único operador: t = -1

>>> u = 2 \*\* 1 \*\* 3

>>> u # qual o valor de u?

Associatividade: u = 2 \*\* 1 \*\* 3 ⇨ u = 2 \*\* 1

Associatividade: u = 2 \*\* 1 ⇨ u = 2

Único operador: u = 2

Note que neste último exemplo foi necessário analisar a associatividade, que no caso do operador de exponenciação é à direita, conforme a Tabela 3.2. O operando 1, por estar entre dois operadores de exponenciação, foi associado àquele à sua direita.

>>> v = ((2 \*\* 1) \*\* 3)

>>> v # qual o valor de v?

Parênteses: v = ((2 \*\* 1) \*\* 3) ⇨ v = (2 \*\* 3)

Parênteses: v = (2 \*\* 3) ⇨ v = 8

Único operador: v = 8

>>> w = (5\*8) + ((9//7) % 2)

>>> w # qual o valor de w?

Parênteses: w = (5\*8) + ((9//7) % 2) ⇨ w = 40 + ((9//7) % 2)

Parênteses: w = 40 + ((9//7) % 2) ⇨ w = 40 + (1 % 2)

Parênteses: w = 40 + (1 % 2) ⇨ w = 40 + 1

Precedência: w = 40 + 1 ⇨ w = 41

Único operador: w = 41

Independente da precedência, a leitura de um código-fonte é feita de cima para baixo, ou seja, da primeira para a última linha de instrução, e cada instrução é lida da esquerda para a direita. Consequentemente, existindo dois pares de parênteses independentes, o mais a esquerda é avaliado primeiro, como visto na Codificação 3.10.

>>> a = b = c = d = 10 # quais os valores das variáveis?

Os operadores de atribuição são não associativos em Python, portanto o encadeamento deles pode gerar uma interpretação especial ou um erro. No caso do operador de atribuição simples (=), há uma interpretação especial, atribuindo o valor mais à direita a todas as variáveis à sua esquerda. Recomenda-se cautela em seu uso, pois pode levar a comportamentos inesperados.

>>> a = b = c = (d = 10) # quais os valores das variáveis?

A Codificação é inválida, pois em Python a atribuição não resulta em valor, logo não há o que atribuir à c. Note que o = entre parênteses está fora do encadeamento.

>>> x = 4

>>> y = 5

>>> x += y += 1 # quais os valores das variáveis?

A Codificação é inválida, pois operadores de atribuição composta são não associativos e quando encadeados geram erro.

**Funções**

Uma função é uma sequência de instruções que executa alguma tarefa específica e que tem um nome. Por exemplo, podemos imaginar uma função que calcula a raiz quadrada de um número não negativo qualquer, ou uma função que coleta dados inseridos pelo usuário, ou uma função que exibe dados na tela.

Podemos dividir as funções em três tipos, de acordo com sua origem:

* **Integradas:** disponibilizadas com a própria linguagem de programação e prontas para uso imediatamente, sem necessidade de instrução adicional;
* **Importadas:** criadas por outros programadores e disponibilizadas para serem incluídas no ambiente de programação, mediante instrução que faça a importação, e então usadas de modo semelhante às integradas;
* **Definidas:** criadas pelo próprio programador e disponíveis para serem utilizadas no código-fonte em que são definidas. É possível distribuí-las para serem importadas em outros códigos-fonte.

Por enquanto focaremos em funções integradas, em próximas aulas criaremos nossas próprias funções que até poderão ser distribuídas para outros programadores!

**Funções integradas**

Uma das vantagens em usar funções integradas, e também das importadas, é a abstração do seu algoritmo, isto é, não precisamos saber detalhes de como foram construídas. O essencial é saber os nomes das funções e o que elas fazem.

Para usar funções integradas, também conhecidas como built-in functions, é necessário chamá-las. Para chamar funções, escreve-se seu nome seguido por um par de parênteses. Ao chamar a função solicitamos sua execução.

>>> print() # chama a função print.

Além de seus nomes e parênteses, a maioria das funções exige um terceiro componente: argumentos. Argumentos são valores inseridos entre os parênteses da chamada e servem para passar dados à função. Os dados passados são necessários para que a função cumpra seu objetivo. Por exemplo, a função abs recebe como argumento um número e retorna o valor absoluto correspondente.

>>> abs(-29) # chama a função abs com o argumento -29.

29 # valor retornado pela função abs.

O valor retornado ou valor devolvido é, como a própria expressão indica, um valor que será dado como retorno pela função, ou seja, a resposta obtida ao chamar a função. Esse valor é o resultado da avaliação da função, algo semelhante ao que ocorre quando avaliamos expressões. Podemos associar os argumentos como as entradas da função e o valor de retorno como a saída da função.

Conceitualmente, nem todas as funções precisam retornar um valor, isso depende do objetivo da função, definido por quem a criou. Não entraremos em todos os detalhes neste momento, mas veremos em outras aulas que, na prática, o Python possui um tratamento especial para funções que não precisariam retornar um valor.

Atente para o fato de que o valor de retorno da função pode ser usado como um valor qualquer, podendo, por exemplo, ser atribuído à uma variável. Veja o exemplo:

>>> x = abs(-29) # atribui o valor de retorno de abs à x.

>>> x + 15

44

Também podemos usar a função em uma expressão com outros operandos, basta lembrar que a função será executada e imaginar que sua chamada será substituída pelo valor retornado, tornando-se um operando como outro qualquer.

>>> abs(-29) + 15 # soma 15 ao valor retornado por abs.

44

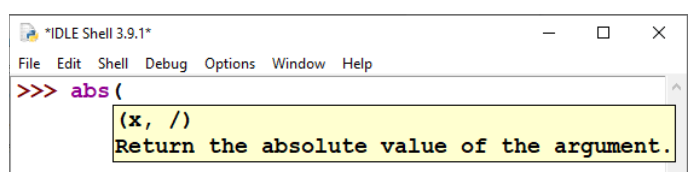
Porém, isso só faz sentido com funções que retornem valor como resposta e que esse valor seja compatível com a expressão onde a função está inserida. Por exemplo, se a função está em uma expressão aritmética, o valor de retorno deve ser um número.

Existem funções que requerem a passagem de mais de um argumento, neste caso eles deverão ser passados na ordem correta e separados por vírgulas. Veja, por exemplo, a função pow que retorna como resposta a potência de uma base elevada a um expoente, semelhante ao operador de exponenciação. Vamos refletir. Para que pow possa realizar a exponenciação, quais os dados necessários? Exatamente! A base e o expoente que se pretende calcular! O primeiro argumento é a base e o segundo é o expoente.

>>> pow(2, 5) # 2 é a base e 5 o expoente

32

Quem determina quais os argumentos e a ordem correta de passá-los à função é seu criador, que pode disponibilizar documentação, geralmente na internet. Também podemos usar o menu Help do IDLE, caso seja uma das funções que acompanham a instalação. A própria Shell e o editor podem ajudar, basta digitar o nome da função com a abertura de parênteses e aguardar alguns instantes.



**Saída de dados**

Você deve ter notado a função embutida print em algumas codificações, apesar de intuitiva por conta do efeito evidente em sua execução, aguardávamos a explicação inicial sobre funções para melhor compreensão. Esta função serve para exibir dados e será bastante usada durante a disciplina.

>>> print('Hello World!') # print com um argumento.

Hello World!

>>> print('2 + 2 =', 2+2) # print com dois argumentos.

2 + 2 = 4

>>> idade = 29

>>> print('Tenho', idade, 'anos') # print com três argumentos.

Tenho 29 anos

Talvez surja a indagação: “por que usar print se ao inserir uma expressão na Shell o resultado é exibido após a execução?”. A Shell do IDLE sempre exibe o resultado de uma expressão após avaliá-la. Porém, isso é uma característica desta ferramenta que foi construída com foco no aprendizado de Python, e não é garantido em outros ambientes, além da Shell ser inadequada para criação de programas completos.

Lembre-se que códigos-fonte sem print, quando executados, não exibem nada.

nome = 'Megan'

nome

idade = 34

idade

Não houve saída? Agora modifique o código-fonte para que fique conforme a Codificação 3.21 e execute-o novamente.

nome = 'Megan'

print(nome)

idade = 34

print(idade)

A função print tem uma característica interessante, pois possui quantidade indeterminada de argumentos, por isso pode ser chamada com zero ou mais argumentos, e o tipo dos argumentos também é arbitrário, podem ser números, strings etc. O Python se encarregará das conversões necessárias para que tudo seja exibido corretamente.

Por padrão, print exibe os valores de seus argumentos e uma quebra de linha '\n', fazendo com que o próximo print exiba seus dados na linha seguinte. Às vezes esse comportamento é indesejado.

print('boa ')

print('noite ')

print('vizinhança')

Se quisermos que apenas o último print quebre a linha, basta acrescentar como último argumento da função um end, seguido de um sinal = com um valor à direita que será exibido no lugar da quebra de linha.

print('boa ', end='')

print('noite ', end='')

print('vizinhança')

No exemplo acima, passamos uma string vazia ('') para o argumento nomeado end, cujo valor padrão é um caractere de quebra de linha ('\n'), não se preocupe em entender os detalhes agora, pois isso será estudado mais a frente, quando aprendermos a definir nossas próprias funções em Python.

A função print não foi projetada para retornar valor, por isso você não a verá como valor de atribuição para uma variável ou usada em uma expressão.

**Entrada de dados**

Para permitir que o usuário do programa forneça dados de entrada, usaremos a função embutida input que, quando executada, faz com que: (I) o programa aguarde um valor de entrada antes de prosseguir com as próximas instruções; (II) converte o valor lido para string e; (III) retorna-o como resposta.

>>> nome = input('Qual o seu nome? ')

Qual o seu nome? Megan

>>> print(nome)

Megan

A função input, pode receber no máximo um argumento, que deve ser uma string e será exibida para o usuário. Note que o retorno de input sempre é uma string e isso pode ser inadequado em algumas situações. Execute no editor a Codificação 3.25, que solicita o nome, o valor do empréstimo e a quantidade de parcelas, e exibe o total da dívida, que é o valor do empréstimo multiplicado pela quantidade de parcelas.

nome = input('Seu nome: ')

valor = input('Valor do empréstimo: ')

parcelas = input('Quantidade de parcelas: ')

print(nome, ', a dívida será de:', valor \* parcelas)

###### **Codificação 3.25: Programa sem conversões dos valores retornados por input.**

Ocorreu um erro, certo? A razão é que as variáveis valor e parcelas receberam strings, consequência do valor retornado por input, e na quarta linha há uma multiplicação entre essas variáveis, ou seja, entre strings, o que é uma expressão inválida. Para resolver esse problema, as entradas deverão ser convertidas para o formato correto, usando as funções integradas int e float.

As funções int e float recebem como argumento um número, ou uma string que represente adequadamente um número do tipo para qual se deseja converter, e devolvem como resposta o valor correspondente convertido para um número inteiro ou em ponto flutuante (real), respectivamente.

nome = input('Seu nome: ')

valor = float(input('Valor do empréstimo: '))

parcelas = int(input('Quantidade de parcelas: '))

print(nome, ', a dívida será de:', valor \* parcelas)

Note que na Codificação 3.26, tanto para a função int quanto para float foi passado como argumento a própria função input, que devolve como resposta uma string. Quando uma função é passada como argumento para outra, a execução inicia pela mais interna (argumento), para que então a mais externa possa ser executada. Vamos analisar a sequência de execução da segunda instrução (input cujo valor de retorno será convertido para float), sendo que o princípio é aplicável também à terceira (input cujo valor de retorno será convertido para int).

Inicialmente input será executada, pois é a função mais interna:

1. valor = float(input('Valor do empréstimo: '))
2. Supondo que o usuário dê a entrada 1000.00, a função input converterá esse valor para string e a retornará como resposta:

valor = float('1000.00')

1. Agora a função float converterá seu argumento, que neste caso é uma string, para o correspondente número real e retornará esse valor.

valor = float('1000.00') ⇨ valor = 1000.0

1. Por fim, o valor 1000.0, de fato um número real representado em ponto flutuante, será atribuído à variável.

valor = 1000.0

**Tipos de erros comuns**

Observamos que erros acontecem! E teremos que lidar com eles, afinal, faz parte da rotina de um programador. Para generalizá-los, classificaremos os erros em três tipos:

* **Erros de sintaxe:** ocorrem por falha na escrita das instruções da linguagem, violando regras e estruturas. Por exemplo, a instrução print 'olá', sem parênteses, é inválida em Python 3, mas é válida em Python 2. Esquecer de completar toda abertura de parênteses com um parêntese de fechamento ou usar um operador binário com apenas um operando são outros exemplos. Geralmente são erros de simples resolução, pois o ambiente de programação costuma indicar instruções com erros de sintaxe ao executar o código;
* **Erros em tempo de execução:** são erros que só aparecem quando o programa é executado, pois a sintaxe está correta. Também podem ser chamados de exceções, indicando que algo excepcional ocorreu, como uma violação do uso esperado do programa. Podem ocorrer, por exemplo, quando o usuário insere um valor inválido ou quando um recurso necessário para a execução do programa está ausente. O exemplo a seguir gera esse tipo de erro:

n = int(input('Número inteiro: ')) # Usuário digita ABC

Caso o usuário digite o que consta no comentário, uma exceção será lançada, indicando que não é possível converter o valor 'ABC' para inteiro.

* **Erros de lógica/semântica:** são erros relacionados ao algoritmo. Nesses casos, o código-fonte pode ser executado e o programa não gerará nenhuma mensagem de erro, mas o resultado não resolve o problema proposto. Algo como “escrever certo a coisa errada”, afinal o algoritmo não soluciona o problema. Estes costumam ser os erros mais difíceis de serem resolvidos.

Podemos resumir os três tipos de erros com a seguinte analogia: (1) em erros de sintaxe, o interpretador não entende o que deve fazer, como falar em um idioma desconhecido; (2) em erros de execução, o interpretador entende o que deve fazer, mas não consegue completar a tarefa, como tentar abrir uma porta com a chave errada; (3) em erros de semântica ou lógica, o interpretador entende o que deve fazer e consegue completar com sucesso todas as instruções, mas isso não resolve o problema especificado, como descarregar um caminhão de tijolos no endereço errado.