

Tutorial Didático de Python

PARTE I – CONCEITOS BÁSICOS



RAFAEL R. PACHECO

SUMÁRIO

Introdução.....	1
CAPÍTULO 1 – Histórico e Preparo.....	2
1.1 Breve Histórico.....	2
1.2 Download e Instalação.....	3
1.3 Integrated Development Environment – IDE.....	3
1.4 Python Enhancement Proposal 8 – PEP 8.....	4
CAPÍTULO 2 – Características e Tipos Básicos.....	5
2.1 Características.....	5
2.1.1 Linguagem de Programação de Alto Nível.....	5
2.1.2 Linguagem Interpretada.....	7
2.1.3 Linguagem Multiparadigma.....	8
2.1.4 Linguagem de Tipagem Dinâmica.....	9
2.1.5 Linguagem Portável.....	9
2.1.6 Linguagem Com Ampla Biblioteca Padrão.....	10
2.2 Tipos Básicos.....	10
2.2.1 Integer.....	11
2.2.2 Float.....	11
2.2.3 String.....	12
2.2.4 Boolean.....	12
2.3 Exemplos.....	13
2.4 Usando TYPE Para Descobrir o Tipo Da Variável.....	13
CAPÍTULO 3 – Operadores, Primeiro Programa e Conversão de Tipos.....	15

3.1 Operadores.....	15
3.1.1 Operador de Atribuição - =.....	15
3.1.2 Operador de Igualdade - ==.....	15
3.1.3 Demais Operadores Relacionais (Comparativos).....	16
3.1.4 Operadores Aritméticos.....	17
3.1.5 Operadores Lógicos.....	18
3.1.7 Outros Operadores.....	19
3.2 Primeiro Programa – Quatro Operações Matemáticas Básicas.....	20
3.2.1 Primeiro Erro.....	21
3.3 Conversão de Tipos – Type Casting.....	23
3.3.1 Conversão Para Inteiro – int().....	24
3.3.2 Conversão Para Float – float().....	25
3.3.3 Conversão Para String – str().....	26
3.4 Melhorando O Programa – Input do Usuário.....	27
 CAPÍTULO 4 – Funções Básicas do Comando print(), Strings Formatadas e Nomes de	
Variáveis.....	35
4.1 Funções Básicas do Comando print().....	35
4.1.1 Exibindo Mais de Um Elemento.....	35
4.1.2 Personalizando O Separador.....	37
4.1.3 Personalizando O Final Da Linha.....	38
4.2 Strings Formatadas.....	39
4.3 Nomes de Variáveis: Regras e Boas Práticas.....	42
4.3.1 Regras Obrigatórias.....	42
4.3.2 Proibições.....	43

4.3.3 Boas Práticas.....	43
CAPÍTULO 5 – Importando Bibliotecas.....	45
5.1 Bibliotecas Padrão.....	45
5.2 Biblioteca vs Módulo.....	45
5.3 Importando – Comando import.....	46
5.3.1 Importando Toda A Biblioteca.....	46
5.3.2 Importando Função Específica da Biblioteca.....	49
5.4 Como Conhecer As Bibliotecas?.....	51
5.4.1 Comando help().....	52
CAPÍTULO 6 – Estruturas de Decisão.....	54
6.1 Definição e Utilidade.....	54
6.2 Estrutura if-else.....	55
6.2.1 Verificando Várias Condições Independentes.....	58
6.2.2 Mais Exemplos de Condições.....	60
6.3 Estrutura if – elif – else.....	62
6.4 Diferenças entre if e elif.....	64
6.5 Estrutura Condicional Aninhada.....	65
CAPÍTULO 7 – Tratamento Básico de Dados Recebidos do Usuário.....	69
7.1 Dados Incorretos.....	69
7.2 Lidando com Espaços – strip().....	69
7.3 Lidando com Maiúsculas e Minúsculas – upper() e lower().....	71
7.4 Enviando Nossas Mensagens de Erro.....	72
7.4.1 Operadores de Pertinência.....	73
7.4.2 Realizando O Tratamento.....	74

7.5 Considerações Finais.....	76
-------------------------------	----

Introdução

Este tutorial foi desenvolvido como projeto de extensão no curso de Ciência da Computação da UNIP, sendo, portanto, totalmente gratuito. O objetivo desse tutorial é ensinar a linguagem de programação Python de forma simples e didática, permitindo que qualquer pessoa possa aprender a programar.

É comum que cursos universitários apenas apresentem noções gerais sobre as linguagens de programação específicas, sem, contudo, se aprofundar em seu ensino. O aluno acaba tendo de aprender por conta própria ou deve buscar outros cursos extracurriculares que ensinem como programar em Python.

Por outro lado, existem diversos cursos muito bons de Python disponíveis no mercado, porém, não raro eles pecam pela didática, especialmente para as pessoas que estão tendo o primeiro contato com a programação.

Este tutorial visa resolver esses problemas: podendo ser utilizado por alunos de cursos voltados à computação ou por qualquer pessoa que deseje aprender a programar em Python, o tutorial passará esses ensinamentos de forma didática e de fácil compreensão, sempre utilizando exemplos.

Esta primeira parte visa apresentar os conceitos introdutórios e básicos para a programação em Python.

Bons estudos!

CAPÍTULO 1 – Histórico e Preparo

1.1 Breve Histórico

A linguagem Python foi desenvolvida pelo programador holandês Guido van Rossum no final dos anos 80 e início dos anos 90. Guido, que utilizava muito a linguagem ABC, a qual ele mesmo havia participado do desenvolvimento, para ensinar novos programadores, almejava criar uma linguagem de programação focada para desenvolvedores profissionais e que fosse tão fácil e intuitiva quanto a ABC.

O nome Python foi uma homenagem ao "Monty Python's Flying Circus", famoso grupo britânico de comédia responsável por vários filmes e sketches televisivos. Guido era fã dessas comédias e julgou que o nome seria apropriado para a nova linguagem que estava desenvolvendo.

Assim, em 1991, Python foi oficialmente lançado na versão 0.9.0 e devido às suas características como facilidade de aprendizado, simplicidade e legibilidade, versatilidade, entre outras, foi ganhando cada vez mais espaço e tornando-se muito popular. Python conta com expressiva e ativa comunidade, que fornece tutoriais, fóruns para discussão e solução de dúvidas e inúmeras bibliotecas para facilitar o desenvolvimento de programas.

O sucesso foi gigantesco: Python é a linguagem de programação mais utilizada atualmente, superando até mesmo JavaScript, Java e C++. Com o tempo, a linguagem foi recebendo atualizações e melhorias. Hoje a versão oficial é o Python 3, sendo que Python 2 foi descontinuado e não mais recebe suporte desde 2020.

Curiosamente, a implementação oficial do Python, chamada CPython, foi escrita em C. Suas funções internas e boa parte da biblioteca padrão também são implementadas em C, o que contribui para uma execução mais eficiente.

1.2 Download e Instalação

Python pode ser instalado nos principais sistemas operacionais, possuindo extrema portabilidade. A linguagem pode rodar em Windows, Linux, MacOS, Android, entre outros sistemas. Para baixar e instalar basta acessar o site oficial - <https://www.python.org/downloads/> - e seguir as instruções de instalação.

Após instalado, o computador já possuirá todos os recursos para dar início ao desenvolvimento de programas em Python e executar arquivos com a extensão .py.

1.3 Integrated Development Environment - IDE

Criar e executar programas em Python é muito simples e pode ser feito inclusive utilizando-se bloco de notas do próprio Windows (ou notas do MacOS) e executar os programas pelo terminal.

Entretanto, existem ferramentas de edição de texto muito práticas e úteis, com diversas funcionalidades desejáveis que irão facilitar a produção, manutenção, correção e execução de código – são as IDEs (Integrated Development Environment), em português, Ambiente de Desenvolvimento Integrado.

Apesar de não serem o foco desse tutorial, recomenda-se ao leitor que baixe e instale uma IDE de sua preferência, pois a sua utilização é praxe do mercado e facilita a produção do desenvolvedor, desde a escrita do código, até a sua formatação e espaçamento de acordo com as normas consideradas como boas práticas.

As IDEs mais utilizadas por desenvolvedores Python são:

Pycharm (Jetbrains);

Visual Studio Code – VSCode (Microsoft);

JupyterLab/Notebook.

VSCode e Pycharm são os mais utilizados. Esse tutorial usará o Pycharm em seus exemplos, pois essa IDE foi projetada especialmente para desenvolvimento de código Python e tem se tornado cada vez mais popular entre as empresas que trabalham com essa linguagem.

1.4 Python Enhancement Proposal 8 – PEP 8

Python Enhancement Proposal 8 é o estilo de escrita de código oficial da linguagem. Trata-se de um manual de boas práticas, formatação e padronização escrito pelo próprio Guido van Rossum, criador do Python, visando tornar a escrita do código mais legível, limpo e consistente.

No decorrer desse tutorial iremos mencionar aspectos do PEP8, tais como nomes de variáveis, tamanho de linhas, espaçamento, dentre outros itens. Sempre que houver dúvida, é recomendável consultar a documentação oficial no site <https://peps.python.org/pep-0008/> (em português: <https://python.org.br/peps/pep-0008/>).

O programador sempre deve se ater às boas práticas, pois facilita não apenas o desenvolvimento do código, mas também a sua compreensão, manutenção, debugging e profissionalismo. A observância do PEP8 é extremamente bem-vista pelas empresas da área tech.

CAPÍTULO 2 – Características e Tipos Básicos

2.1 Características

Python é uma linguagem de programação de alto nível, interpretada, multiparadigma, de tipagem dinâmica, portátil, com ampla biblioteca padrão e de sintaxe simples. Passaremos à análise de cada uma dessas principais características isoladamente.

Vamos explicar esses conceitos um a um. É compreensível que no começo haja certa dificuldade em entender todos eles, pois dependem de conhecimentos que somente serão adquiridos em capítulos posteriores. O importante nesse momento é apenas ter noção dessas características principais do Python.

2.1.1 Linguagem de Programação de Alto Nível

Classificar Python como linguagem de alto nível significa dizer que a sua sintaxe - a forma como o código é escrito e as funções são chamadas - é muito próximo da linguagem humana. Programar em Python é muito parecido com escrever um texto em inglês.

Isso traz enormes vantagens para aprender a linguagem, escrever o código e entender o que cada programa faz e como faz. O próprio Python já cuida de questões abrir e fechar espaços na memória, alocar informações e valores das variáveis e outras funções sem que o programador tenha de se preocupar com elas.

Em contrapartida, linguagens de baixo nível são mais próximas da linguagem da máquina. O computador não consegue entender a nossa língua, convertendo tudo para o sistema binário – zeros e uns (0 e 1). Programar em linguagem de baixo nível representa um desafio muito maior, pois a compreensão é severamente afetada pela maior complexidade do código.

Como exemplo, a doutrina costuma citar e comparar Python (alto nível) com Assembly (baixo nível), criando um programa em Linux que exiba no console a frase “Hello, World!”.

Esse programa ficaria dessa maneira em Assembly utilizando Linux como Sistema Operacional:

```
section .data

    msg db 'Hello, World!', 0xA ; Mensagem + nova linha

    len equ $ - msg           ; Comprimento da mensagem

section .text

    global _start

_start:

    ; syscall: write(fd=1, buf=msg, len)

    mov eax, 4    ; syscall número 4: sys_write

    mov ebx, 1    ; file descriptor 1: saída padrão (stdout)

    mov ecx, msg  ; ponteiro para a mensagem

    mov edx, len  ; comprimento da mensagem

    int 0x80      ; interrupção para chamada de sistema

    ; syscall: exit(status=0)

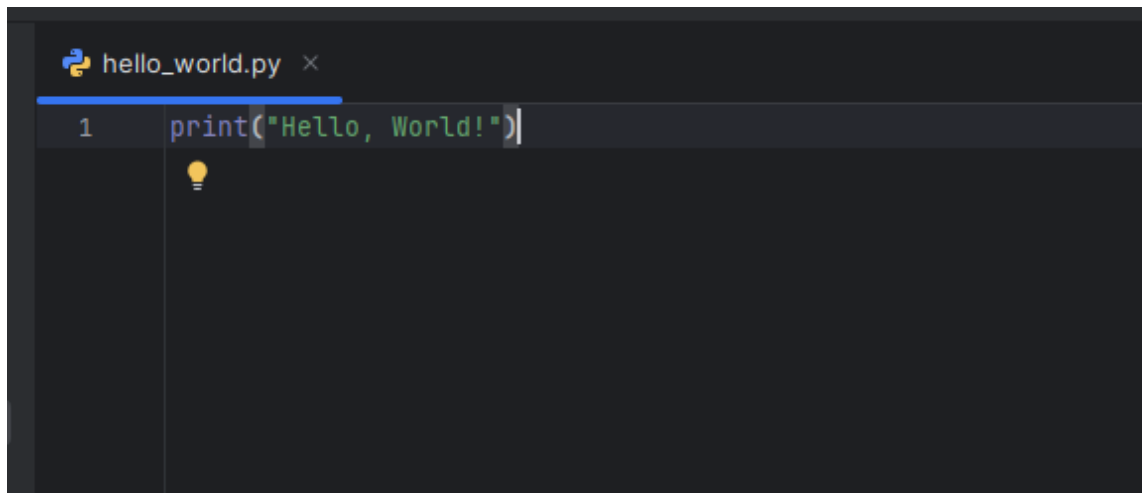
    mov eax, 1    ; syscall número 1: sys_exit

    xor ebx, ebx  ; status = 0

    int 0x80
```

Como Podemos ver, o código é complexo e envolve até o controle da memória sobre a alocação, uso e liberação.

O mesmo programa utilizando Python é extremamente simples:

A screenshot of a code editor window titled 'hello_world.py'. The editor has a dark background. On line 1, the code 'print("Hello, World!")' is written in a light green font. A lightbulb icon is visible below the code line, indicating a suggestion or tip. The cursor is at the end of the line.

Em apenas uma linha conseguimos obter o mesmo resultado. O próprio Python cuida do controle de memória e o programador somente precisou utilizar a função `print()` e colocar o texto desejado.

“Embora as linguagens de baixo nível sejam mais rápidas e eficientes em termos de desempenho, as facilidades, a legibilidade e a produtividade oferecidas pelas linguagens de alto nível — como Python — fazem com que elas sejam amplamente preferidas no desenvolvimento de software moderno.”

2.1.2 Linguagem Interpretada

Linguagens interpretadas são lidas e executadas linha por linha por meio de um interpretador. O computador somente entende a linguagem de máquina (código binário de 0s e 1s) e os programas escritos em Python precisam ser traduzidos para que o computador possa entender o programa.

O interpretador irá ler e traduzir cada linha de código. As principais vantagens de ser interpretada consistem em:

- a) Execução é imediata e não precisa criar um arquivo compilado (como .exe no Windows ou .app em Mac);
- b) O programa pode ser executado em qualquer computador com o Python instalado, independente do Sistema Operacional;

Já a principal desvantagem é que os programas são mais lentos do que aqueles que usam um arquivo compilado para a leitura do código, pois esse arquivo compilado já estará em linguagem de máquina ou em linguagem mais próxima da forma binária.

2.1.3 Linguagem Multiparadigma

Paradigmas de linguagens são abordagens e estilos de programação que priorizam determinadas formas e estruturas para gerar um código limpo, consistente e robusto. Há vários paradigmas, sendo que os principais e mais utilizados atualmente são:

- a) Estruturado – o programa é feito como uma sequência de instruções (condicionais, laços e funções);
- b) Programação Orientada a Objetos – o programa é organizado com objetos (classes);
- c) Programação Orientada a Eventos – o programa roda aguardando eventos externos (como input do usuário) para executar determinadas ações.

Não iremos explicar todos os paradigmas de linguagem, pois não é o foco desse tutorial. No momento basta sabermos que o Python possui essa característica de suportar múltiplos paradigmas, sendo que falaremos um pouco mais sobre eles quando for conveniente.

2.1.4 Linguagem de Tipagem Dinâmica


O Python consegue inferir qual seria o tipo da variável quando ela é declarada. Se o programador define o valor 30 para uma variável chamada idade, o próprio Python já vai entender que se trata de um número inteiro. Da mesma forma, se o programador cria uma variável chamada nome e define o valor “José” para ela, o Python já entende que isso é uma string (um texto).

Linguagens que não são dinâmicas (chamadas estáticas) não permitem isso. Antes de declarar a variável e atribuir valor a elas, o programador precisa explicitamente escrever qual será o tipo dela. Veja o exemplo em Java:

```
int idade = 30;
```

O programador precisou usar a palavra “int” para dizer ao programa que 30 deve ser considerado um número inteiro.

Já em Python:

```
1  idade = 30 # o próprio Python já sabe que 30 é um número inteiro!  
2  
```

2.1.5 Linguagem Portável

Os programas em Python podem rodar em qualquer máquina que tenha o Python instalado, independente do Sistema Operacional. Isso significa que os arquivos de extensão .py podem ser lidos por uma máquina Linux mesmo que o programador tenha feito o código usando o Windows (ou qualquer outro Sistema Operacional), por exemplo.

Isso torna essa linguagem extremamente versátil, pois não é preciso reescrever código nem se preocupar em criar um arquivo compilado (vantagem de ser interpretada, conforme item 2.1.2) para que ela rode em qualquer computador. Basta ter o Python instalado.

2.1.6 Linguagem Com Ampla Biblioteca Padrão

Uma das grandes vantagens do Python é que ele possui uma ampla biblioteca padrão, conhecida pela expressão “batteries included” (pilhas incluídas). Isso significa que a linguagem já vem, por padrão, com uma enorme quantidade de módulos e ferramentas prontas para uso, sem que o programador precise instalar bibliotecas externas.

Esses módulos cobrem uma variedade de tarefas comuns do dia a dia da programação, como:

- a) Módulo math para funções matemáticas;
- b) Módulo datetime para datas e horários;
- c) Módulo random para gerar números aleatórios;
- d) Módulo os para acessar e manipular o sistema operacional;

Existem muitas outras bibliotecas padrão que acompanham o Python. Essas bibliotecas são extremamente úteis, pois já vêm com funções prontas para serem utilizadas. O programador não precisa programar tudo do zero, podendo fazer uso das bibliotecas.

Por exemplo, caso precise que o programa acesse a data do computador no momento em que ele está sendo executado, o programador não precisa escrever do zero código novo para isso, bastando utilizar as funções do módulo datetime.

2.2 Tipos Básicos

Também chamados de tipos primitivos, são as categoriais fundamentais de dados. Toda vez que atribuímos algum valor para determinada variável, esse valor é classificado em um desses tipos básicos.

A definição do tipo é importante, pois determina como o dado poderá ser manipulado, o que pode e não pode ser feito com ele. Em Python, nós temos os seguintes tipos básicos (ou tipos primitivos):

- a) Integer (número inteiro) - int;
- b) Float (número com casa decimal) - float;
- c) String (texto) - str;
- d) Boolean (True e False – Verdadeiro e Falso) - bool;

Além disso, também há o valor nulo, que representa justamente a ausência de valor – None, porém, conforme veremos futuramente, o valor nulo não possui métodos específicos, servindo apenas para informar que não há valor definido para aquela variável.

2.2.1 Integer

Representa o tipo inteiro para valores numéricos. Pode trabalhar com números negativos ou positivos e o 0, contanto que não tenham casas decimais. Por exemplo: 10; -70; 30; 42; 98; 0.

Todos esses números são valores inteiros e serão armazenados como integer em Python. Internamente, o Python chama esse tipo de int, sendo que quando formos programar iremos várias vezes nos deparar com essa expressão.

2.2.2 Float

Representa o tipo decimal para valores numéricos. Também pode trabalhar com números negativos ou positivos e o 0. Atenção que para o Python não se deve usar vírgula (,) para separar a parte inteira da decimal, mas sim o ponto (.), igual os americanos fazem. Então se quisermos representar “1,5”, teremos de escrever como “1.5” para não ocorrer um erro.

Além disso, é possível que a casa decimal seja zero. O número 10.0 será considerado como float, ainda que a casa decimal seja 0. Veja exemplos de float: 3.1415; -78.32; 0.0; -10.0; 86435.456.

Todos esses números são valores com casa decimal e serão armazenados como float em Python. Internamente, o Python chama esse tipo de float.

2.2.3 String

Representa o tipo texto ou sequência de caracteres. A string sempre deve ser indicada entre aspas simples ('TEXTO') ou duplas ("TEXTO") e pode ser vazia (não conter nada, somente as aspas – ""), ter apenas um caractere (exemplo: "a"), apenas um espaço (" ") ou todo um conjunto de caracteres, de tamanho indefinido. Por exemplo, se atribuirmos a uma variável o texto "Estou me esforçando muito para estudar Python!", toda essa frase será considerada uma string.

Atenção que não é o conteúdo do texto que determina que ele será uma string. Se atribuirmos a uma variável o valor "3", o Python vai entender isso como uma string. Mesmo que 3 seja um número inteiro, para o Python as aspas vão indicar que o programador não deseja usar esse dado como número, mas sim um texto com o caractere "3". Isso também vale para "5.60" (valores decimais entre aspas) e até mesmo para "True" e "False" (valores booleanos entre aspas).

Internamente, o Python chama esse tipo de str.

2.2.4 Boolean

Representa o tipo verdadeiro ou falso – True or False (sempre com letra maiúscula para o Python). É um tipo binário que somente armazena esse tipo de informação – verdade ou falsidade. Não é possível trabalhar com meio termo: ou a variável recebe True ou recebe False.

Internamente, o Python chama esse tipo de bool.

2.3 Exemplos

```
1  idade = 30 # a variável tem valor int
2
3  nome = "André" # a variável tem valor str
4
5  conta = 72.80 # a variável tem valor float
6
7  temperatura = 10.0 # a variável tem valor float
8
9  resultado = -15 # a variável tem valor int
10
11 tres = "3" # a variável tem valor str
12
13 nota = 0.0 # a variável tem valor float
14
15 alugado = True # a variável tem valor bool
16
17 texto = "" # a variável tem valor str
18
19 desafio = "False" # a variável tem valor str
20
21 numero = -78.0792 # a variável tem valor float
```

2.4 Usando TYPE Para Descobrir o Tipo Da Variável

É possível descobrir a qual tipo primitivo uma variável pertence usando `type()` junto com `print()`. A função `type()` vai retornar o tipo da variável e a função `print()`, como já vimos no exemplo do item 2.1.1, irá imprimir o resultado no console.

O método que usaremos será o seguinte: escrever `print(type(variável))`. São duas etapas:

- a) `type()` com a variável dentro = `type(variável)`;

b) colocar `type(variável)` dentro de `print()` = `print(type(variável))`

Usando as mesmas variáveis que declaramos e atribuímos valor para dar o exemplo, teremos os seguintes resultados:

```
1  idade = 30 # a variável tem valor int
2  nome = "André" # a variável tem valor str
3  conta = 72.80 # a variável tem valor float
4  temperatura = 10.0 # a variável tem valor float
5  resultado = -15 # a variável tem valor int
6  tres = "3" # a variável tem valor str
7  nota = 0.0 # a variável tem valor float
8  alugado = True # a variável tem valor bool
9  texto = "" # a variável tem valor str
10 desafio = "False" # a variável tem valor str
11 numero = -78.0792 # a variável tem valor float
12
13 print(type(idade))
14 print(type(nome))
15 print(type(conta))
16 print(type(temperatura))
17 print(type(resultado))
18 print(type(tres))
19 print(type(nota))
20 print(type(alugado))
21 print(type(texto))
22 print(type(desafio))
23 print(type(numero))
```

```
<class 'int'>
<class 'str'>
<class 'float'>
<class 'float'>
<class 'int'>
<class 'str'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'str'>
<class 'float'>
```

Sempre que estiver em dúvida use `print()` em conjunto com `type()` e execute o programa para que seja exibido o tipo básico da variável no terminal.

CAPÍTULO 3 – Operadores, Primeiro Programa e Conversão de Tipos

3.1 Operadores

Os operadores são símbolos e palavras reservadas utilizados para realizar operações entre os valores, denominados “operandos”. São essenciais para a elaboração dos programas e o programador deve ter familiaridade com todos, pois são usados com enorme frequência.

3.1.1 Operador de Atribuição - =

Em Python, o sinal de igualdade (=) é utilizado para atribuir valores a uma variável. Quando o programador escreve uma linha de código dessa forma:

```
1  idade = 30
2
```

Ele está iniciando uma variável chamada idade e está atribuindo o valor 30 a ela. Atenção que esse sinal não é usado para verificar igualdade entre duas coisas, mas sim para atribuir valores. No exemplo acima da variável idade, devemos ler a linha de código da seguinte maneira: “idade **RECEBE** 30”.

Não estamos falando que idade é igual a 30, mas sim que ela recebeu o valor inteiro 30. Essa sutileza pode parecer boba, porém é importante criar o hábito de prestar atenção nela para evitar confusões futuramente.

3.1.2 Operador de Igualdade - ==

Para verificarmos a existência de igualdade entre dois operandos, devemos utilizar dois sinais de igualdades em seguida e sem espaço (==). Se as variáveis que estiverem sendo comparadas forem iguais, o programa retornará True (verdadeiro). Caso contrário, o retorno será False (falso).

Continuando com o exemplo da idade, vamos realizar as comparações e determinar a sua impressão no console usando print():

```
operadores.py ×  
1  idade = 30  
2  
3  print(idade == 25)  
4  print(idade == 30)  
5
```

Resultado no terminal:

```
False  
True  
  
Process finished with exit code 0
```

Observe que o primeiro print exibiu False, pois idade recebeu o valor 30 na linha 1 e 30 não é igual a 25. Por outro lado, o segundo print retornou True, pois idade vale 30 e 30 é igual a 30.

3.1.3 Demais Operadores Relacionais (Comparativos)

Além do operador de igualdade, outros comparativos são:

- a) diferença: !=
- b) maior que: >
- c) menor que: <
- d) maior ou igual a: >=
- e) menor ou igual a: <=

Montemos uma tabela com os operadores relacionais e alguns exemplos:

Operador	Significado	Exemplo
==	Igual a	$3 == 2 \rightarrow \text{False}$
!=	Diferente de	$3 != 2 \rightarrow \text{True}$
>	Maior que	$3 > 2 \rightarrow \text{True}$
<	Menor que	$3 < 2 \rightarrow \text{False}$
>=	Maior ou igual a	$3 >= 2 \rightarrow \text{True}$
<=	Menor ou igual a	$3 <= 2 \rightarrow \text{False}$

3.1.4 Operadores Aritméticos

Esses operadores realizam as operações matemáticas básicas. São eles:

- a) adição: +
- b) subtração: -
- c) multiplicação: *
- d) divisão: /
- e) divisão inteira: //
- f) módulo (resto da divisão): %
- g) exponenciação: **

Montando uma tabela:

Operador	Significado	Exemplo
+	Adição	$3 + 2 = 5$
-	Subtração	$3 - 2 = 1$
*	Multiplicação	$3 * 2 = 6$
/	Divisão	$3 / 2 = 1.5$
//	Divisão Inteira	$3 // 2 = 1$
%	Módulo	$3 \% 2 = 1$
**	Exponenciação	$3 ** 2 = 9$

Muita atenção para a diferença entre a divisão, a divisão inteira e o módulo.

A divisão vai retornar a divisão comum e completa que estamos acostumados, mostrando um número fracionário, caso seja necessário.

A divisão inteira, por sua vez, somente irá mostrar a parte inteira. Veja no exemplo: se 3 for dividido por 2, o resultado é 1.5 (lembrando que no Python usamos ponto e não vírgula para separar a casa decimal). Na divisão inteira, a parte fracionária (0.5) será desprezada e por isso teremos 1 (inteiro) como resultado.

O módulo segue lógica parecida, mas em vez de exibir o resultado da divisão, ele exibe o resto que não foi dividido. No exemplo de 3 dividido por 2, o número 2 somente cabe uma vez dentro de 3, de forma que para continuar a divisão e pegar a parte decimal, teríamos de fazer 1 dividido por 2. Esse 1 que sobrou é justamente o resto que o operador módulo irá exibir.

3.1.5 Operadores Lógicos

Esses operadores servem para combinar expressões booleanas (aquelas que são True ou False). Em Python nós temos três operadores lógicos:

- a) and
- b) or
- c) not

Começemos pelo operador AND, chamado de E lógico. Esse operador combina as expressões lógicas usando o conectivo “e”. **Ele somente retornará True se todas as expressões combinadas também forem True. Se somente uma das expressões for False, o resultado todo será False.**

```
1 resultado = True and True and False and True
2
3 print(resultado)
4
```

No console:

```
False

Process finished with exit code 0
```

O **operador OR**, por sua vez, é o OU lógico. Esse operador combina as expressões lógicas usando o conectivo “ou”. **Basta que uma das expressões seja True que ele retornará True. Ele somente retornará False se todas as expressões forem False.**

```
1 resultado = True or True or False or True
2
3 print(resultado)
4 |
```

No console:

```
True

Process finished with exit code 0
```

Por fim, o **operador NOT** é a negação. Esse operador inverte o valor da expressão lógica. Então **not True** retornará **False** e **not False** retornará **True**.

3.1.7 Outros Operadores

Existem, ainda, outros operadores que consistem em:

- a) operadores de identidade;
- b) operadores de pertinência;
- c) operadores bit a bit (bitwise)

No entanto, esses operadores não serão estudados no momento, pois poderão confundir o aprendizado. Futuramente falaremos mais sobre eles, pois é mais fácil de entender os seus usos quando o estudante já elaborou alguns programas e já tem certa noção de como é programar em Python.

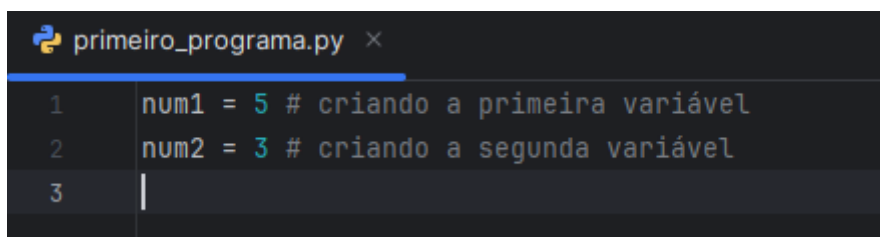
Por enquanto, basta saber que eles existem e que os dois primeiros (identidade e pertinência) são muito utilizados.

3.2 Primeiro Programa – Quatro Operações Matemáticas Básicas

Agora que já sabemos como criar variáveis usando o operador de atribuição e também conhecemos os operadores aritméticos, podemos montar um programa simples que receba dois números e exibe o resultado da soma, da subtração, da multiplicação e da divisão.

Para isso basta criarmos duas variáveis diferentes e atribuir valores a elas. Logo em seguida podemos utilizar o `print()` para exibir o resultado das operações. Vejamos parte a parte:

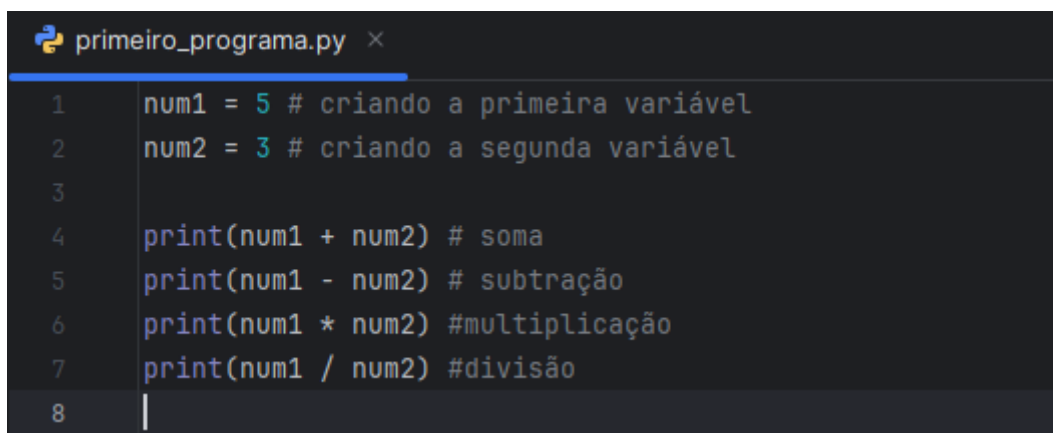
Criando as variáveis:



```
primeiro_programa.py x
1 num1 = 5 # criando a primeira variável
2 num2 = 3 # criando a segunda variável
3
```

Criamos a primeira variável chamada `num1` que recebeu 5. Na segunda linha criamos a `num2` que recebeu 3. Agora vamos printar os resultados das operações.

Printando as Operações:



```
primeiro_programa.py x
1 num1 = 5 # criando a primeira variável
2 num2 = 3 # criando a segunda variável
3
4 print(num1 + num2) # soma
5 print(num1 - num2) # subtração
6 print(num1 * num2) #multiplicação
7 print(num1 / num2) #divisão
8
```

Quando executarmos o programa, as operações serão exibidas no console. Ficará dessa forma:

```
8
2
15
1.6666666666666667
```

```
Process finished with exit code 0
```

Outra forma de lidar com a questão consiste em passar o resultado das operações para outras variáveis e depois utilizar o comando `print()` com essas variáveis criadas. O programa fica assim:

```
primeiro_programa.py x
1  num1 = 5 # criando a primeira variável
2  num2 = 3 # criando a segunda variável
3
4  resultado_soma = num1 + num2 # atribuímos o resultado da soma para uma nova variável
5  resultado_subtracao = num1 - num2 # atribuímos o resultado da subtração para uma nova variável
6  resultado_multiplicacao = num1 * num2 # atribuímos o resultado da multiplicação para uma nova variável
7  resultado_divisao = num1 / num2 # atribuímos o resultado da divisão para uma nova variável
8
9
10 print(resultado_soma)
11 print(resultado_subtracao)
12 print(resultado_multiplicacao)
13 print(resultado_divisao)
14
```

O resultado impresso no console é idêntico ao exemplo anterior, pois fizemos a mesma coisa:

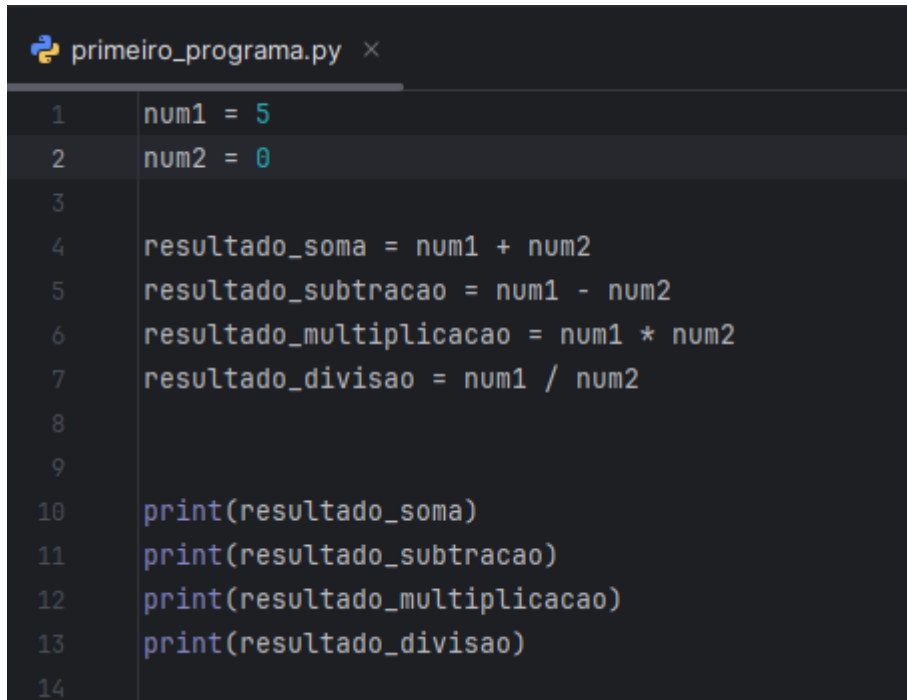
```
8
2
15
1.6666666666666667
```

```
Process finished with exit code 0
```

3.2.1 Primeiro Erro

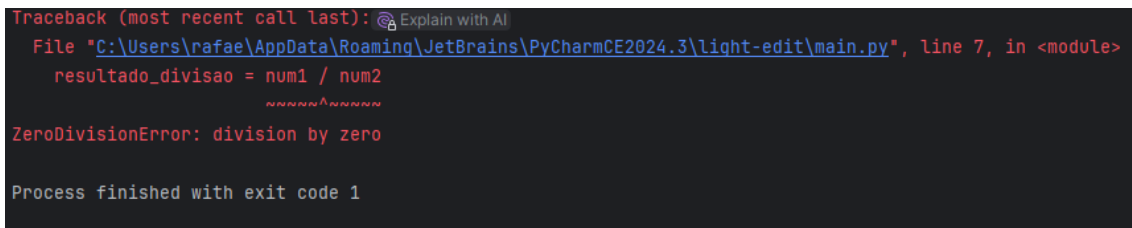
Vamos aproveitar esse programa para termos contato com a nossa primeira mensagem de erro. Dentre as operações matemáticas que o nosso programa realiza temos a divisão. E se atribuirmos o valor 0 à variável `num2`? Conseguiremos fazer uma divisão por 0? A matemática não proíbe essa operação? Vamos tentar!

Removemos todos os comentários do código e mudamos o valor de num2 para 0. O código ficou assim:



```
primeiro_programa.py x
1 num1 = 5
2 num2 = 0
3
4 resultado_soma = num1 + num2
5 resultado_subtracao = num1 - num2
6 resultado_multiplicacao = num1 * num2
7 resultado_divisao = num1 / num2
8
9
10 print(resultado_soma)
11 print(resultado_subtracao)
12 print(resultado_multiplicacao)
13 print(resultado_divisao)
14
```

Ao executar, aparece a seguinte mensagem no console:



```
Traceback (most recent call last):
  File "C:\Users\rafae\AppData\Roaming\JetBrains\PyCharmCE2024.3\light-edit\main.py", line 7, in <module>
    resultado_divisao = num1 / num2
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Vamos entender essa mensagem de erro.

O Python explica que encontrou um erro no arquivo executado, demonstrando o seu caminho e especificando o lugar exato que o erro aconteceu.

No caso a mensagem vermelha indica que o erro foi na linha 7 do nosso programa e replica o código que deu erro: `resultado_divisao = num1 / num2`.

Ao final, é explicado qual foi o erro: `ZeroDivisionError: Division by zero`.

Ou seja, foi encontrado um erro na linha 7 do nosso código. Nessa linha foi identificada uma tentativa de divisão por 0, algo que é impossível fazer. O programador deve ficar muito atento às mensagens de erro, pois elas indicarão o que aconteceu e onde está o problema. Isso facilita muito a correção e ajuste do código para resolver eventuais equívocos.

Observe que o erro quebrou o programa. Nada mais foi executado. As demais operações que estavam corretas e sem erro não foram exibidas porque a ocorrência do erro tem precedência sobre todo o resto do programa. Assim que o erro é detectado, nada mais do código será executado e o Python exibirá quais erros foram encontrados e em quais linhas do código eles estão.

Existem formas de tratar erros e evitar que eles interrompam completamente a execução do programa, permitindo que ele continue mesmo após a ocorrência de um problema. No entanto, essas técnicas serão explicadas em um capítulo posterior.

Por enquanto, o mais importante é entender que, em Python, todo e qualquer erro não tratado fará com que o programa seja interrompido imediatamente, e será exibida uma mensagem de erro no console, informando o tipo do erro e a linha de código onde ele ocorreu.

3.3 Conversão de Tipos – Type Casting

O Python permite que convertamos o tipo de uma variável para outro tipo. Podemos, por exemplo, transformar o tipo inteiro 10 em uma string “10”. Esse é o chamado Type Casting – conversão de tipos.

Essa funcionalidade é muito útil, pois há casos em que a tipagem dinâmica do Python não é suficiente para resolver todas as necessidades do programa.

Para realizar a conversão basta usar a função construtora do tipo para o qual queremos converter o valor e fornecer a variável que passará pela conversão. As funções construtoras são:

- a) `int()` para converter para número inteiro;
- b) `float()` para converter para número com casa decimal;
- c) `str()` para converter para string;
- d) `bool()` para converter para booleano

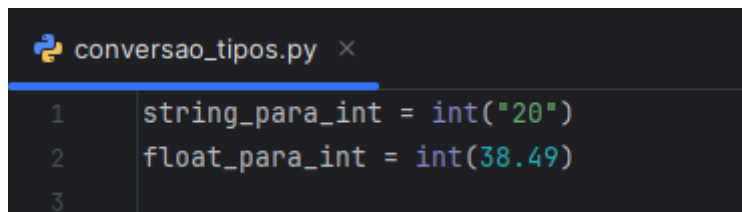
Vamos explicar quais são as conversões possíveis, detalhando as regras e usando exemplos para facilitar a compreensão.

3.3.1 Conversão Para Inteiro – `int()`

Esse tipo de conversão pode ser usado para:

- a) converter strings **numéricas** para inteiro;
- b) converter valores decimais para inteiro, **com perda da parte decimal**.

Para converter o valor de uma variável para o número inteiro, basta usar a função `int(valor a ser convertido)`. Veja:



```
conversao_tipos.py x
1 string_para_int = int("20")
2 float_para_int = int(38.49)
3
```

Neste exemplo, a variável `string_para_int` recebeu o valor 20 (inteiro), pois a função `int()` converteu “20” que era string para 20, valor numérico inteiro.

Já a variável `float_para_int` recebeu o valor 38 (inteiro, com desprezo da casa decimal), pois a função `int()` converteu o valor decimal 38.49 para 38, valor inteiro. Observe que houve perda da parte decimal (0.49 foi perdido e desprezado).

Executando os comandos `print(string_para_int)` e `print(float_para_int)` temos os seguintes resultados:

```
20
38

Process finished with exit code 0
```

Essas são as únicas conversões possíveis para `int`. Caso tentemos fazer outra conversão teremos erro. Vamos exemplificar tentando converter “dez” (string) para `int` e printar a variável que recebeu a conversão:

```
conversao_tipos.py x
1 conversao_impossivel = int("dez")
2
3 print(conversao_impossivel)
4 |
```

Executando o programa:

```
Traceback (most recent call last):
  File "C:\Users\rafae\PycharmProjects\Tutorial\.venv\conversao_tipos.py", line 1, in <module>
    conversao_impossivel = int("dez")
ValueError: invalid literal for int() with base 10: 'dez'

Process finished with exit code 1
```

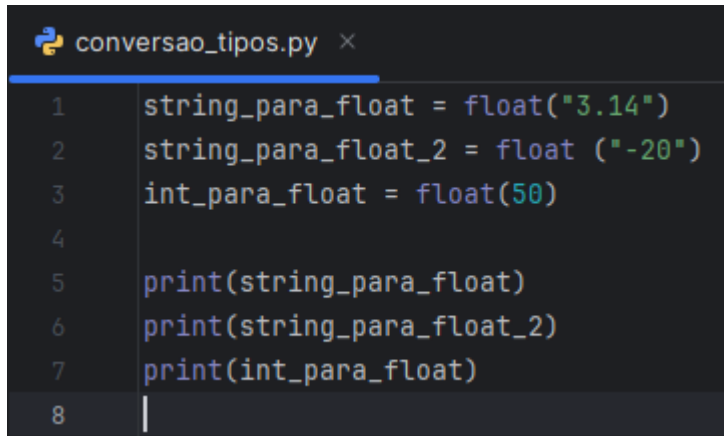
Recebemos um erro de valor explicando que não há como converter a string “dez” para um número de base 10. Isso se dá porque “dez” não é uma string numérica. Exemplos de strings numéricas: “10”; “70.13”; “-15”; “0”.

3.3.2 Conversão Para Float – `float()`

Esse tipo de conversão pode ser usado para:

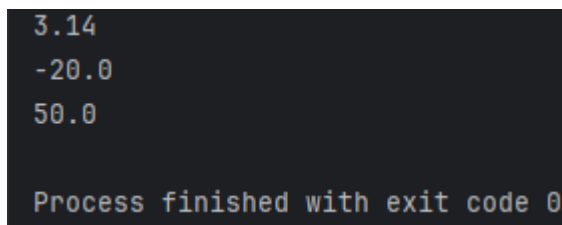
- a) converter strings **numéricas** para decimal;
- b) converter valores inteiros para decimal, **com ganho de casa decimal**.

Para converter o valor de uma variável para o número decimal, basta usar a função `float(valor a ser convertido)`. Veja:



```
conversao_tipos.py x
1 string_para_float = float("3.14")
2 string_para_float_2 = float("-20")
3 int_para_float = float(50)
4
5 print(string_para_float)
6 print(string_para_float_2)
7 print(int_para_float)
8
```

Resultado dos prints no console:



```
3.14
-20.0
50.0

Process finished with exit code 0
```

Observe que os valores que não tinham casa decimal, no caso a string “20” e o inteiro 50, ganharam .0 como casa decimal, tornando-se, respectivamente, 20.0 e 50.0, valores float.

Assim como ocorre na conversão para inteiro, não há como converter strings não numéricas (“dez”, por exemplo) para um número decimal, de forma que essa conversão é impossível e também gera erro de valor.

3.3.3 Conversão Para String – `str()`

Esse tipo de conversão pode ser usado para:

- a) converter qualquer tipo de valor para uma string (transforma em texto).

Essa é a forma de conversão mais ampla que o Python permite, pois qualquer tipo de valor pode ser convertido para string. Números inteiros, números decimais e até mesmo valores booleanos podem ser convertidos. Exemplos:

```
conversao_tipos.py ×  
1 int_para_string = str(27)  
2 float_para_string = str(-20.89)  
3 bool_para_string = str(True)  
4  
5 print(int_para_string)  
6 print(float_para_string)  
7 print(bool_para_string)  
8
```

Resultado dos prints no console:

```
27  
-20.89  
True  
  
Process finished with exit code 0
```

Como podemos ver, todos os valores foram convertidos com sucesso para strings, e passaram a ser tratados como texto.

3.4 Melhorando O Programa – Input do Usuário

Podemos melhorar o programa permitindo que o usuário passe quais números que ele quer usar para as operações matemáticas. Até agora, se quisermos mudar os números nós temos de alterar o valor das variáveis no próprio código, mas o Python permite que deixemos essa definição a cargo de quem está usando o programa.

Para isso temos de usar a função `input()`. Ela irá permitir que o usuário digite o valor que ele deseja que as variáveis recebam. Vamos explicar como funciona.

Em vez de atribuir um valor diretamente à variável, o programador atribui a função `input()` à essa variável, junto com uma mensagem de texto (string). Veja o exemplo:

```
primeiro_programa.py x
1 num1 = input("Digite o primeiro número: ")
2 num2 = input("Digite o segundo número: ")
3
```

Observe que agora as variáveis `num1` e `num2` não mais estão recebendo o valor, mas sim a função `input()` com uma mensagem junto. Não alteramos mais nada no programa e somente modificamos as variáveis `num1` e `num2`. Vejamos como ficou:

```
1 num1 = (input("Digite o primeiro número: "))
2 num2 = (input("Digite o segundo número: "))
3
4 resultado_soma = num1 + num2
5 resultado_subtracao = num1 - num2
6 resultado_multiplicacao = num1 * num2
7 resultado_divisao = num1 / num2
8
9
10 print(f"Foram digitados os números {num1} e {num2}.")
11 print(f"O resultado da soma entre eles é: {resultado_soma}")
12 print(f"O resultado da subtração entre eles é: {resultado_subtracao}")
13 print(f"O resultado da multiplicação entre eles é: {resultado_multiplicacao}")
14 print(f"O resultado da divisão entre eles é: {resultado_divisao}")
15
```

Quando rodarmos o programa, a função `input()` irá funcionar e as mensagens aparecerão no terminal para o usuário, num prompt com espaço para digitar o valor que deseja passar. Isso será feito primeiro para o `num1` e depois para o `num2`, obedecendo a ordem em que eles aparecem.

Rodando o programa, acontece o seguinte no console:

```
Digite o primeiro número:
```

Vamos digitar 10 e confirmar com a tecla `enter`:

```
Digite o primeiro número: 10
```

Após confirmar, num1 receberá 10 e o programa irá pedir o valor de num2:

```
Digite o primeiro número: 10
Digite o segundo número:
```

Agora vamos colocar o valor 2 para o segundo número e vamos confirmar com o enter:

```
Digite o primeiro número: 10
Digite o segundo número: 2
```

Assim que confirmar o programa vai continuar, sendo que num1 recebeu 10 e num2 recebeu 2, conforme determinado pelo usuário. Vamos ver o resultado:

```
Digite o primeiro número: 10
Digite o segundo número: 2
Traceback (most recent call last):
  File "C:\Users\rafae\PycharmProjects\Tutorial\.venv\primeiro_programa.py", line 5, in <module>
    resultado_subtracao = num1 - num2
                          ~~~~~^~~~~~
TypeError: unsupported operand type(s) for -: 'str' and 'str'

Process finished with exit code 1
```

Recebemos uma mensagem de erro! O que aconteceu?

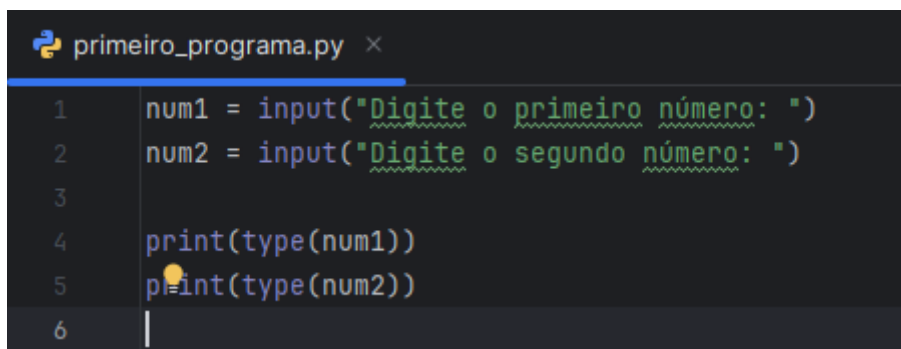
O Python está apontando que o erro está na linha 5 quando tentamos realizar a subtração num1 – num2. Ao tentar fazer essa operação ele detectou um erro de tipo e informou que esse erro aconteceu porque estamos tentando realizar a subtração de duas strings (str).

Espere! Mas não digitamos dois números? 10 e 2? Por que o Python está dizendo que temos duas strings em vez de dois números inteiros?

Isso acontece porque o que o usuário digita é considerado como texto. A função `input()` passa o valor para a variável na forma de string (texto) por padrão. Então, na verdade, `num1` não recebeu 10 (número inteiro 10), mas sim “10” (string. Texto contendo “10”).

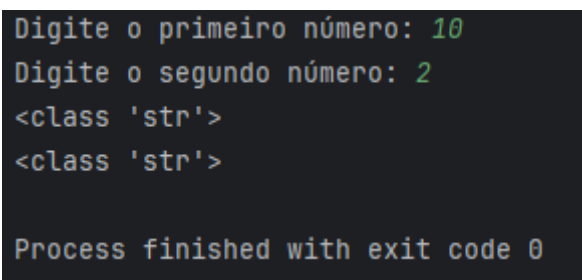
Da mesma forma, `num2` não recebeu 2 (número inteiro 2), mas sim “2” (string. Texto contendo “2”).

Apagando todo o resto do programa e usando em conjunto as funções `print()` e `type()` – aprendemos isso no capítulo 2, item 2.4), não resta dúvida de que `num1` e `num2` receberam strings:



```
primeiro_programa.py x
1 num1 = input("Digite o primeiro número: ")
2 num2 = input("Digite o segundo número: ")
3
4 print(type(num1))
5 print(type(num2))
6
```

Resultado impresso no console após inserirmos os mesmos valores (10 e 2):



```
Digite o primeiro número: 10
Digite o segundo número: 2
<class 'str'>
<class 'str'>

Process finished with exit code 0
```

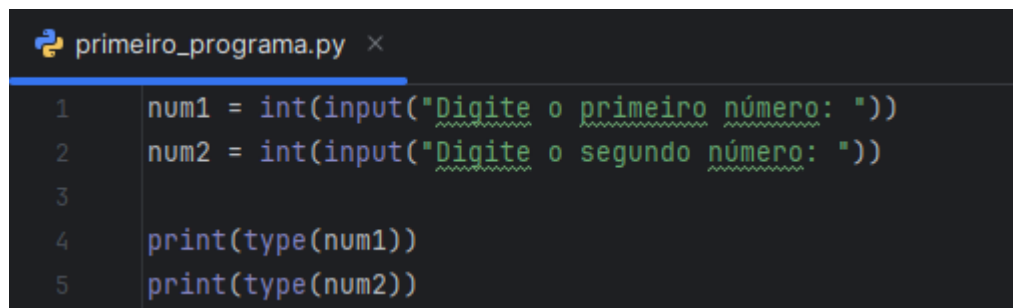
As variáveis `num1` e `num2` receberam strings então!

Agora que constatamos o “problema”, podemos pensar na solução: temos de converter o valor string para um valor numérico. No caso do nosso programa, pode ser tanto `int` (inteiro) quanto `float` (decimal).

Vamos converter para inteiro (int)! Já sabemos como fazer isso, pois no item 3.3.1 aprendemos sobre a conversão de string para inteiro (str para int) usando o type casting. Podemos fazer isso de duas formas diferentes:

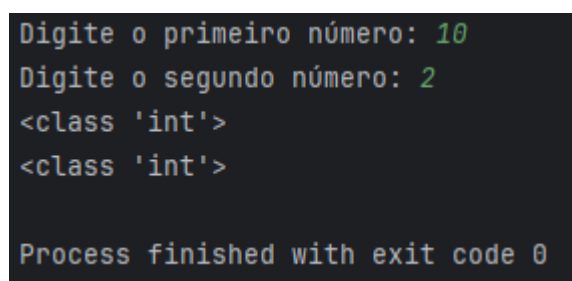
- a) já realizar a conversão junto do próprio comando de input;
- b) converter os valores recebidos no input e armazenar em outras variáveis.

Para realizarmos no próprio input, basta inserir todo o comando de input dentro do int(), da mesma forma que fizemos nos exemplos do item 3.3.1:



```
primeiro_programa.py ×
1 num1 = int(input("Digite o primeiro número: "))
2 num2 = int(input("Digite o segundo número: "))
3
4 print(type(num1))
5 print(type(num2))
```

Agora todo o comando de input está dentro da função int(). O Python irá receber o valor do usuário e automaticamente irá convertê-lo para inteiro. Vejamos o que o console exibirá como resultado dos prints quando executarmos o programa, considerando que o usuário novamente inseriu os números 10 e 2 nos inputs:



```
Digite o primeiro número: 10
Digite o segundo número: 2
<class 'int'>
<class 'int'>

Process finished with exit code 0
```

Ambas as variáveis num1 e num2 agora são números inteiros! O programa rodará corretamente. Vamos testar:

```
primeiro_programa.py ×
1  num1 = int(input("Digite o primeiro número: "))
2  num2 = int(input("Digite o segundo número: "))
3
4  resultado_soma = num1 + num2
5  resultado_subtracao = num1 - num2
6  resultado_multiplicacao = num1 * num2
7  resultado_divisao = num1 / num2
8
9
10 print(resultado_soma)
11 print(resultado_subtracao)
12 print(resultado_multiplicacao)
13 print(resultado_divisao)
14
```

No console:

```
Digite o primeiro número: 10
Digite o segundo número: 2
12
8
20
5.0

Process finished with exit code 0
```

Conseguimos! Não recebemos mais a mensagem de erro que indicava ser impossível subtrairmos duas strings. O programa recebeu números inteiros nas variáveis e todas as operações matemáticas foram realizadas corretamente!

Para converter os valores e armazenar em outras variáveis, nós iremos manter os comandos de input e vamos criar duas novas variáveis que receberão o valor convertido por meio do type casting usando `int()`. Também é simples:

```
primeiro_programa.py x
1 num1 = input("Digite o primeiro número: ")
2 num2 = input("Digite o segundo número: ")
3
4 num1_convertido = int(num1)
5 num2_convertido = int(num2)
```

Criamos as variáveis `num1_convertido` e `num2_convertido` e elas receberam, respectivamente, os valores de `num1` e `num2` e os converteram para inteiro. Para o programa funcionar, agora devemos garantir que as operações matemáticas usarão as novas variáveis, pois `num1` e `num2` ainda são strings. Veja o exemplo:

```
primeiro_programa.py x
1 num1 = input("Digite o primeiro número: ")
2 num2 = input("Digite o segundo número: ")
3
4 num1_convertido = int(num1)
5 num2_convertido = int(num2)
6
7 resultado_soma = num1_convertido + num2_convertido
8 resultado_subtracao = num1_convertido - num2_convertido
9 resultado_multiplicacao = num1_convertido * num2_convertido
10 resultado_divisao = num1_convertido / num2_convertido
11
12
13 print(resultado_soma)
14 print(resultado_subtracao)
15 print(resultado_multiplicacao)
16 print(resultado_divisao)
17
```

Tivemos de usar as novas variáveis. O resultado das operações será o mesmo do exemplo acima no qual convertemos diretamente no input:

```
Digite o primeiro número: 10
Digite o segundo número: 2
12
8
20
5.0
Process finished with exit code 0
```

Nosso programa também funcionou corretamente.

ATENÇÃO: provamos que ambos os métodos funcionam, mas temos algumas diferenças entre eles:

- converter direto no input faz com que as variáveis num1 e num2 já recebam valores do tipo inteiro (int);

- converter usando novas variáveis faz com que as variáveis num1 e num2 recebam valores do tipo string (str). As novas variáveis num1_convertido e num2_convertido é que receberam os valores como inteiro (int);

- **CONCLUSÃO:** no segundo método terminamos com 4 variáveis que podem ser usadas: num1 e num2 são strings e num1_convertido e num2_convertido são inteiros.

ATENÇÃO 2: Caso o usuário digite um valor que não possa ser convertido para inteiro, o programa irá quebrar e exibir erro de valor, informando a impossibilidade de realizar a conversão. Esse tipo de problema pode ser tratado, mas não é conveniente estudarmos isso agora, pois é um tópico mais avançado. O importante agora é saber que esse erro pode acontecer.

CAPÍTULO 4 – Funções Básicas do Comando print(), Strings Formatadas e Nomes de Variáveis

4.1 Funções Básicas do Comando print()

Já usamos o comando `print()` para exibir mensagens, conteúdo de variáveis e resultados de expressões no console. Agora que estamos acostumados com o seu uso e sabemos como ele se comporta, podemos aprender algumas de suas principais funcionalidades e capacidades de personalização.

4.1.1 Exibindo Mais de Um Elemento

Em todos os exemplos que usamos, sempre demos o comando `print()` com apenas um parâmetro. Sempre printamos de forma individual uma mensagem, ou o resultado de uma expressão, ou o valor de uma variável.

Entretanto, isso não é obrigatório. O Python permite que passemos diversos parâmetros para a função `print()` dentro da mesma linha de código. Para exibir mais de um elemento, basta usar o `print()` e inserir os elementos que desejamos exibir no console, utilizando uma vírgula (,) para separá-los e indicar que estamos enviando mais de um parâmetro.

Usando essa técnica, todos os elementos que mandamos printar serão exibidos no console em uma única linha na mesma ordem que foram inseridos no `print()` e serão separados por um espaço em branco.

Vamos usar o programa das quatro operações matemáticas básicas que criamos no capítulo anterior para exemplificar. Usando essa função do `print()` iremos deixar a visualização das operações mais fácil para o usuário, indicando qual operação foi realizada antes de cada resultado.


```
primeiro_programa.py x
1  num1 = int(input("Digite o primeiro número: "))
2  num2 = int(input("Digite o segundo número: "))
3
4  resultado_soma = num1 + num2
5  resultado_subtracao = num1 - num2
6  resultado_multiplicacao = num1 * num2
7  resultado_divisao = num1 / num2
8
9
10 print("Soma:", resultado_soma)
11 print("Subtração:", resultado_subtracao)
12 print("Multiplicação:", resultado_multiplicacao)
13 print("Divisão:", resultado_divisao)
14
```

Mantivemos o mesmo programa e apenas alteramos os comandos `print()` aplicando o que aprendemos. Observe que inserimos o texto que queremos exibir como primeiro parâmetro e, após usar a vírgula, inserimos a variável desejada.

Resultado no console:

```
Digite o primeiro número: 10
Digite o segundo número: 2
Soma: 12
Subtração: 8
Multiplicação: 20
Divisão: 5.0

Process finished with exit code 0
```

Agora o programa está explicando qual operação foi feita para chegar em cada valor, facilitando a compreensão do usuário.

O `print()` aceita que façamos qualquer combinação e que enviemos qualquer número de parâmetros. Podemos mandar três strings separadas por vírgula, ou um valor booleano e outras quatro variáveis. Seguem alguns exemplos:

```

1  variavel_1 = "Nome"
2  variavel_2 = "Sobrenome"
3
4  print("Somente números:", 1, 4, -3.7)
5  print("Somente variáveis:", variavel_1, variavel_2)
6  print("Texto e variável:", "Qual é o seu", variavel_1, "?")
7  print("Variável e boolean:", variavel_1, True)
8

```

Resultado:

```

Somente números: 1 4 -3.7
Somente variáveis: Nome Sobrenome
Texto e variável: Qual é o seu Nome ?
Variável e boolean: Nome True

Process finished with exit code 0

```

4.1.2 Personalizando O Separador

Conforme vimos nos exemplos acima, o separador padrão do `print()` é um espaço em branco, igual o que usamos para separar as palavras quando escrevemos. O Python permite que editemos e personalizemos esse separador.

Podemos aplicar o separador personalizado que desejarmos passando o comando `sep = "separador"` como último parâmetro do `print()`. Como exemplo usaremos o sinal de hífen (-).

```

1  print("Printando", "múltiplos", "elementos", "com", "separador", "personalizado", sep = "-")
2  print("Printando", "múltiplos", "elementos", "com", "separador", "padrão")
3

```

No console:

```

Printando-múltiplos-elementos-com-separador-personalizado
Printando múltiplos elementos com separador padrão

Process finished with exit code 0

```

No primeiro caso as palavras foram separadas pelo hífen, pois passamos o parâmetro `sep = "-"` no final, indicando para a função `print()` que desejamos usar esse

traço como separador. Qualquer caractere pode ser usado como separador. O programador é quem escolhe!

Já o segundo print foi utilizado com o separador padrão, então não foi necessário passar nenhum outro parâmetro para o print().

4.1.3 Personalizando O Final Da Linha

A função print() também permite que escolhamos o que será o final da linha. Em todos os exemplos que demos até agora, podemos notar que o print() sempre pulou uma linha. Todas as sequências de prints nossas sempre foram exibidas no console linha a linha, print a print.

Isso acontece porque o padrão de fim de linha para o print() é justamente pular uma linha. No Python, o comando para pular linha é \n. Toda vez que o programa encontra um \n ele sabe que o programador deseja que ele pule essa linha.

Como o \n é o padrão do print(), assim como acontece com o espaço no separador, o programador não precisa fazer nada quando não quer modificar esse comportamento.

Por outro lado, se houver a necessidade de personalizar o final da linha, o programador pode passar o comando end = "final desejado" como último parâmetro enviado para a função print(), exatamente como fizemos com o separador.

Vamos impedir que ele pule a linha passando um espaço (" ") como parâmetro:

```

1 print("Eu", end=" ")
2 print("não", end=" ")
3 print("quero", end=" ")
4 print("pular", end=" ")
5 print("linha")
6

```

Atenção que o último print() não precisa que não pulemos a linha, pois é o último print() da nossa sequência. Veja o resultado no console:

```

Eu não quero pular linha
Process finished with exit code 0

```

Nenhuma linha foi pulada e o espaço foi usado como fim da linha. Resultado: a frase saiu como se tivesse sido passada print("Eu não quero pular linha") para o programa. Assim como no caso do separador, o programador tem ampla liberdade para escolher como finalizar a linha. Qualquer caractere pode ser passado como fim de linha.

4.2 Strings Formatadas

O Python permite que formatemos strings para criar um texto que use o valor de variáveis sem ter de recorrer ao método que explicamos acima, usando vírgulas para separar os parâmetros.

Essas strings formatadas são conhecidas como f strings e a sua sintaxe é: f'texto {variável}' e são extremamente úteis e importantes para a programação. Para formatar a string começamos com a letra f, pois isso indica ao Python que desejamos fazer uso dessa função.

Posteriormente, escrevemos a string normalmente (o texto que desejamos). Quando precisarmos inserir no texto o valor de determinada variável, basta colocar essa variável dentro de chaves ({}) no próprio texto.

Com o exemplo fica fácil entender. Novamente usaremos o nosso programa das quatro operações matemáticas básicas. Relembrando o programa e o resultado no console:

```
primeiro_programa.py x
1  num1 = int(input("Digite o primeiro número: "))
2  num2 = int(input("Digite o segundo número: "))
3
4  resultado_soma = num1 + num2
5  resultado_subtracao = num1 - num2
6  resultado_multiplicacao = num1 * num2
7  resultado_divisao = num1 / num2
8
9
10 print("Soma:", resultado_soma)
11 print("Subtração:", resultado_subtracao)
12 print("Multiplicação:", resultado_multiplicacao)
13 print("Divisão:", resultado_divisao)
14
```

```
Digite o primeiro número: 10
Digite o segundo número: 2
Soma: 12
Subtração: 8
Multiplicação: 20
Divisão: 5.0

Process finished with exit code 0
```

Vamos usar strings formatadas em cada comando `print()` para que seja exibido no console dois tipos de textos diferentes:

- o primeiro tipo de texto irá informar quais números o usuário passou ao programa;
- o segundo tipo de texto mostrará a operação matemática realizada e o resultado. Um texto para cada operação feita;

Usando a sintaxe que aprendemos – `f'texto {variável}'`:

```

1  num1 = int(input("Digite o primeiro número: "))
2  num2 = int(input("Digite o segundo número: "))
3
4  resultado_soma = num1 + num2
5  resultado_subtracao = num1 - num2
6  resultado_multiplicacao = num1 * num2
7  resultado_divisao = num1 / num2
8
9
10 print(f"Foram digitados os números {num1} e {num2}.")
11 print(f"O resultado da soma entre eles é: {resultado_soma}")
12 print(f"O resultado da subtração entre eles é: {resultado_subtracao}")
13 print(f"O resultado da multiplicação entre eles é: {resultado_multiplicacao}")
14 print(f"O resultado da divisão entre eles é: {resultado_divisao}")
15

```

Não precisamos mais usar as vírgulas, pois agora os valores das variáveis estão dentro da própria string formatada. Vejamos o resultado ao rodar o programa:

```

Digite o primeiro número: 10
Digite o segundo número: 2
Foram digitados os números 10 e 2.
O resultado da soma entre eles é: 12
O resultado da subtração entre eles é: 8
O resultado da multiplicação entre eles é: 20
O resultado da divisão entre eles é: 5.0

Process finished with exit code 0

```

F strings são importantíssimas e o programador fará uso delas com grande frequência. Pratique até ficar completamente confortável com a sua sintaxe.

ATENÇÃO: Muito cuidado com as aspas. Sabemos que podemos escrever as strings tanto entre aspas simples ('texto') quanto entre aspas duplas ("texto"). Caso precise usar uma palavra entre aspas **dentro** de uma string formatada, use aspas de tipo diferente daquelas usadas para formatar a string.

Exemplo usando aspas duplas para string e aspas simples para um termo dentro dela:

```
1 print(f"Como usei aspas duplas para a f string agora escreverei 'texto' com aspas simples!")
2
```

Resultado:

```
Como usei aspas duplas para a f string agora escreverei 'texto' com aspas simples!
Process finished with exit code 0
```

Fazendo dessa forma nós evitamos erro de sintaxe e o programa funciona corretamente.

4.3 Nomes de Variáveis: Regras e Boas Práticas

Quando criamos variáveis há regras e boas práticas a serem seguidas, bem como nomes a serem evitados. Escolher bem o nome das variáveis ajuda na compreensão de sua função no programa, facilitando não apenas a elaboração do código, mas também na sua manutenção e eventual correção de erros e bugs. A clareza sempre é bem-vinda.

4.3.1 Regras Obrigatórias

São regras consideradas obrigatórias pela documentação do Python. São elas:

- a) nomes de variáveis devem começar com letra minúscula ou _;
- b) nomes de variáveis podem conter números e _

Exemplos:

```
nomes_variaveis.py x
1 nome = "José"
2 _idade = 30
3 nome_completo = "Machado de Assis"
4 nome2 = "Maria"
5
```

Essas são as regras obrigatórias e sempre devem ser seguidas. O Python até aceita que o programador quebre essas regras, mas isso é considerado extrema falta de profissionalismo e pode, em alguns casos, quebrar o código.

4.3.2 Proibições

O programador nunca deve utilizar espaços no início do nome da variável e nem usar para separar palavras em caso de nome composto. Não escreva nome completo, mas sim `nome_completo`.

Não use pontuação, como vírgula (,), ponto final (.) ou ponto e vírgula (;).

É proibido usar palavras reservadas pela linguagem e os nomes de funções. Não nomeie um atributo como `int` ou `print`. `Int = 5` e `print = "texto"` são proibidos, pois estão usando palavras reservadas (`int`) e nome de uma função embutida (`print`) como nome de variáveis. Isso irá causar grandes problemas no código e, na maioria dos casos, quebrará o programa.

4.3.3 Boas Práticas

Além de respeitar as regras obrigatórias e as proibições, o programador deve se atentar para as seguintes boas práticas:

- a) não iniciar nome da variável com número (exemplo: `2valor = False`);
- b) atribuir nome que seja útil para entender o que é e para que serve a variável;
- c) usar underlines (`_`) para separar palavras ou snakeCase (convenção que dita que a primeira palavra começa com letra minúscula e a segunda começa com maiúscula, sem espaço entre elas);
- d) escrever o nome de CONSTANTES em caixa alta (exemplo: `PI = 3.1415`).

ATENÇÃO: algumas linguagens de programação permitem que o programador crie constantes em vez de variáveis. Uma vez definida como constante, não há mais uma “variável”, mas sim um valor que não pode ser mudado no decorrer do código. **O PYTHON NÃO TEM SUPORTE PARA A CRIAÇÃO DE CONSTANTES.**

Como não há esse suporte, a comunidade Python criou a convenção de nomear a variável em caixa alta, ou seja, letra maiúscula, para informar aos demais programadores que aquela variável não deve ser alterada.

Assim, toda vez que nos depararmos com uma variável escrita com letras maiúsculas, sabemos que quem fez o programa está nos avisando para não alterarmos o valor daquela variável!

CAPÍTULO 5 – Importando Bibliotecas

5.1 Bibliotecas Padrão

O Python possui vasta quantidade de bibliotecas que acompanham a linguagem de programação por padrão, conforme explicado no item 2.1.6 (batteries included). Conforme já mencionamos, essas bibliotecas trazem funcionalidades novas que permitem ao programador utilizá-las sem ter de programar tudo do zero.

O módulo `datetime` é excelente para ilustrarmos a utilidade de importar funções da biblioteca padrão. Suponha que o programador precise que determinado aplicativo exiba a data e hora em que o programa foi rodado.

Para isso, ele teria de escrever um código que fosse capaz de interagir com o computador para pegar os dados necessários (data e hora) e exibi-los. Isso demandaria tempo e conhecimentos técnicos de como fazer isso.

Acontece que a biblioteca do `datetime`, que já vem por padrão no Python, tem uma função específica para fazer isso! O programador do nosso exemplo não precisa escrever todo o código do zero, bastando importar o módulo `datetime` e usar essa função.

5.2 Biblioteca vs Módulo

Os termos biblioteca e módulo são intercambiáveis. Na prática, usamos como sinônimos. Porém, vale a pena ressaltar que a teoria cita uma pequena diferença entre elas: a biblioteca é uma pasta com vários módulos. O módulo é um arquivo específico.

Assim, um módulo é um único arquivo com extensão `.py` que pode ser importado e uma biblioteca é um conjunto de módulos organizados como pacote.

Futuramente trataremos melhor desse assunto, especificamente quando formos criar nossos próprios pacotes e módulos em vez de usar os que já vêm como padrão. Por enquanto apenas entenda que os termos “biblioteca” e “módulo” tem uma distinção doutrinária (acadêmica), mas que na prática são usados como sinônimos.

Para facilitar, também usaremos esses termos como sinônimos.

5.3 Importando – Comando import

Para importarmos uma biblioteca padrão, basta usarmos o comando import que pode ser utilizado de duas formas:

- a) importar toda a biblioteca, com todas as funções que ela possui;
- b) importar somente uma função da biblioteca.

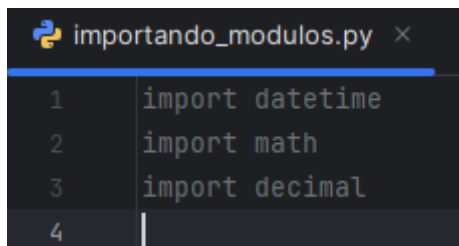
A escolha do tipo de import que faremos é importante, pois determina como será a sintaxe do comando e como será a sintaxe para usar a função desejada. Vamos explicar tudo detalhadamente.

5.3.1 Importando Toda A Biblioteca

Para importar a biblioteca como um todo, basta escrever a seguinte linha de código: `import nome da biblioteca`.

Mencionamos o `datetime` acima. Então no código escrevemos: `import datetime`. Essa linha de código deve ser a primeira coisa no programa, pois é uma regra de boas práticas.

Então todos os imports que formos realizar deverão ser a primeira coisa que faremos no código. Vamos supor que o nosso programa utilize três bibliotecas diferentes, a saber, `datetime`, `math` e `decimal`. Faremos da seguinte forma:

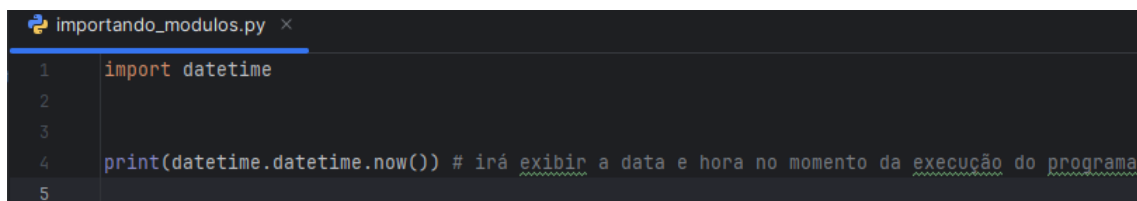


```
importando_modulos.py x
1 import datetime
2 import math
3 import decimal
4
```

Com essas linhas de código essas três bibliotecas foram importadas em sua totalidade, ou seja, todas as suas funções estão disponíveis para uso pelo programa. As palavras estão em cinza porque é a forma do Pycharm (a IDE utilizada) informar que ainda não usamos nenhuma das funcionalidades dessas bibliotecas importadas no programa.

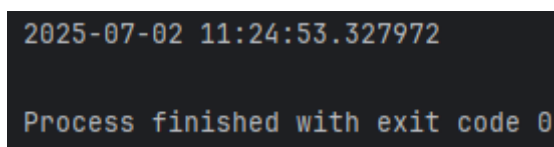
Vamos usar a datetime como exemplo. Uma de suas funções é obter a data e hora do computador do usuário e permitir que armazenemos essa informação em uma variável, ou que a exibamos diretamente no console usando o comando `print()`. A função que permite isso tem o nome de `datetime.now()`.

A sintaxe para usar uma função de uma biblioteca importada em sua totalidade é: `nomedabiblioteca.funcao()`. No caso específico do nosso exemplo, ficaria `datetime.datetime.now()`:



```
importando_modulos.py x
1 import datetime
2
3
4 print(datetime.datetime.now()) # irá exibir a data e hora no momento da execução do programa
5
```

Executando:



```
2025-07-02 11:24:53.327972
Process finished with exit code 0
```

Foram exibidos a data e o horário no formato ano – mês – dia e hora:minuto:segundo, inclusive com milésimos de segundo.

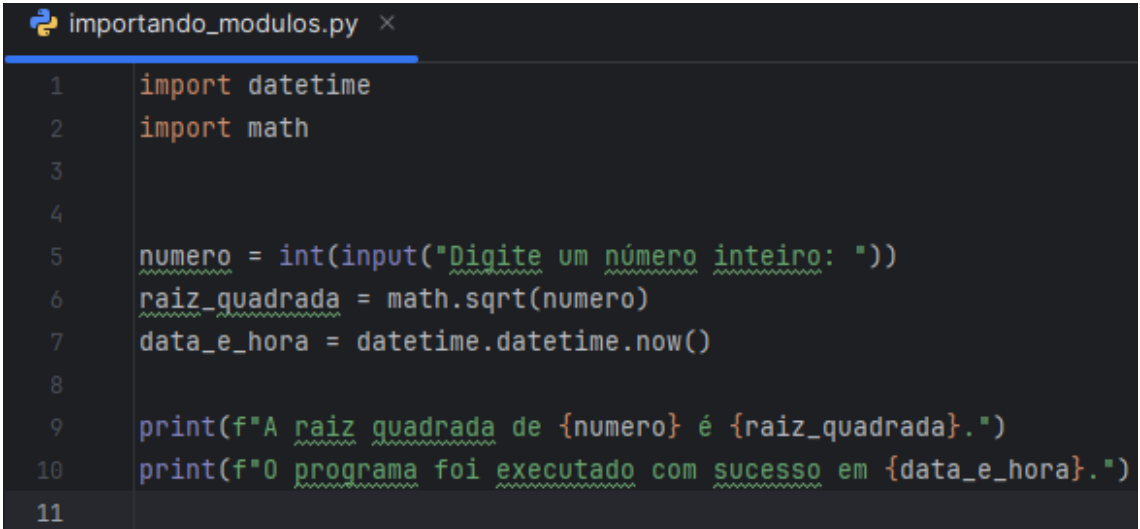
Usando bibliotecas, vamos criar agora um programa que recebe um valor inteiro do usuário e exibe a raiz quadrada desse número, bem como a data e hora em que o programa foi executado.

Para exibirmos a raiz quadrada usaremos o módulo math, com a função chamada sqrt().

O programa deve seguir essas etapas:

- importar as bibliotecas necessárias;
- receber input do usuário;
- armazenar a raiz quadrada do número em uma variável;
- armazenar os dados datetime em outra variável;
- exibir no console usando prints com strings formatadas (f strings).

Escrevendo o código:



```
importando_modulos.py x
1  import datetime
2  import math
3
4
5  numero = int(input("Digite um número inteiro: "))
6  raiz_quadrada = math.sqrt(numero)
7  data_e_hora = datetime.datetime.now()
8
9  print(f"A raiz quadrada de {numero} é {raiz_quadrada}.")
10 print(f"O programa foi executado com sucesso em {data_e_hora}.")
11
```

Resultado no console:

```
Digite um número inteiro: 5
A raiz quadrada de 5 é 2.23606797749979.
0 programa foi executado com sucesso em 2025-07-02 11:33:27.642785.

Process finished with exit code 0
```

A utilização de módulos facilita muito a vida do programador. Com poucas linhas de código conseguimos executar funções complexas que demandariam conhecimento técnico e levariam tempo para serem corretamente implementadas e testadas.

5.3.2 Importando Função Específica da Biblioteca

O Python também permite que importemos somente as funções que vamos usar. Isso tem a vantagem de tornar o programa mais leve, mas o programador já deve saber exatamente quais funções de cada biblioteca ele irá precisar.

A sintaxe desse tipo de import é simples: devemos usar a palavra reservada `from` da seguinte forma: `from biblioteca import função`. Lembrando que qualquer tipo de import sempre deve ser a primeira coisa a se fazer no programa, conforme determinam as boas práticas.

Vamos usar como exemplo a função `randint()` da biblioteca `random` para simularmos o lançamento de um dado de vinte faces, igual usamos nos jogos de *Dungeons and Dragons*.

A função `randint()` funciona assim: passamos a ela dois números inteiros separados por vírgula e ele sorteará de forma pseudo-aleatória um número dentro desse intervalo.

Importando a função:

```
importando_modulos.py x
1 from random import randint
2
```

A função já foi importada. Agora podemos utilizá-la. Quando importamos uma função dessa maneira, a sintaxe para chamá-la é um pouco mais simples do que aquela do exemplo do import do módulo datetime como um todo.

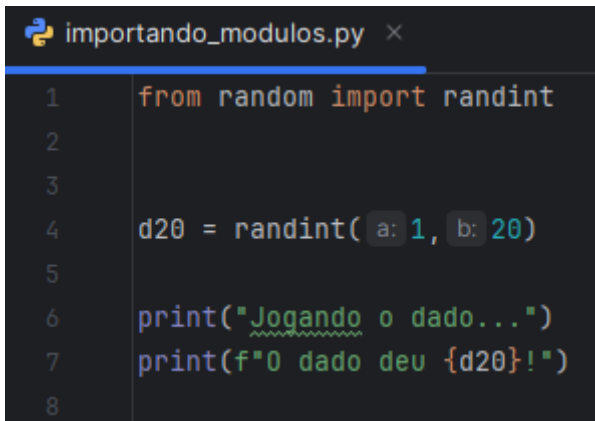
Quando importamos a função diretamente e de forma isolada, nós a chamamos diretamente pelo nome, sem ter que especificar o nome da biblioteca antes. Comparando as duas formas para facilitar a compreensão:

```
importando_modulos.py x
1 from random import randint
2 import math
3
4
5 d20 = randint(a: 1, b: 20)
6 raiz_quadrada = math.sqrt(2)
```

A variável d20 está recebendo o resultado da função randint() do módulo random com intervalo de sorteio entre 1 e 20. Veja que não foi preciso escrever random.randint(), pois ela foi importada usando a fórmula from random import randint.

Por outro lado, a variável raiz_quadrada está recebendo o resultado da função sqrt do módulo math para calcular a raiz quadrada de 2. Observe que fomos obrigados a escrever math.sqrt(), pois ela foi importada usando a fórmula import math.

Vamos continuar com o nosso programa que joga o dado de 20 faces:

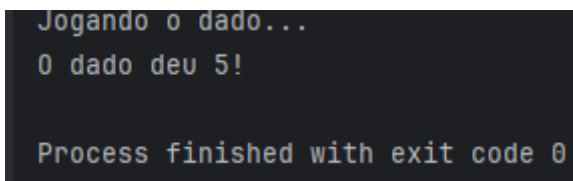


```

1  from random import randint
2
3
4  d20 = randint(a=1, b=20)
5
6  print("Jogando o dado...")
7  print(f"O dado deu {d20}!")
8

```

Resultado:



```

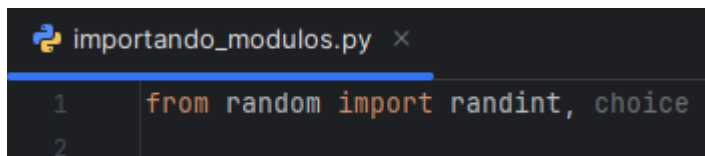
Jogando o dado...
O dado deu 5!

Process finished with exit code 0

```

A própria função `randint()` é que escolheu 5 de forma pseudo-aleatória.

ATENÇÃO: É possível importar mais de uma função na mesma linha de código usando esse método. Para fazer isso, basta colocar o nome das funções que deseja importar separados por vírgulas (,). Exemplo:



```

1  from random import randint, choice
2

```

Importamos da biblioteca `random` as funções `randint()` e `choice()`!

5.4 Como Conhecer As Bibliotecas?

Conforme mencionado diversas vezes, existem muitas bibliotecas (módulos) que acompanham o Python de forma padrão. Todas elas possuem ferramentas que auxiliam muito o trabalho do programador.

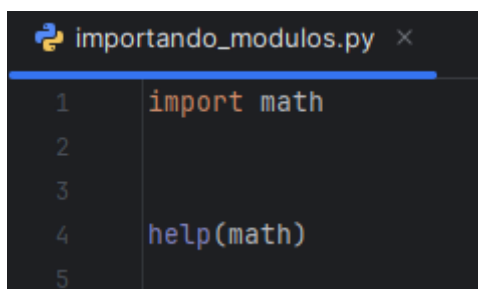
Para saber quais são as bibliotecas built-in do Python é necessário ler a documentação da linguagem. Além disso, cada uma das bibliotecas terá a sua própria documentação explicando quais são as suas funcionalidades e como devem ser usadas.

Existem bibliotecas criadas pela comunidade que não acompanham o Python, ou seja, não são built-in, mas podem ser usadas mediante download e instalação. O programador também pode criar as suas próprias bibliotecas. Estudaremos melhor esses processos em outro capítulo. Por enquanto, apenas saiba que existem inúmeras bibliotecas que podem auxiliar o programador.

5.4.1 Comando help()

O Python possui um comando chamado help() que ajuda a entender o que uma biblioteca pode oferecer. Basta digitar help(biblioteca) que ele irá exibir a documentação da biblioteca importada. Vamos ver o exemplo com o módulo math. Para isso precisaremos fazer o seguinte:

- a) importar o módulo math;
- b) usar o comando help(math)



```
importando_modulos.py x
1  import math
2
3
4  help(math)
5
```

O resultado no console será muito extenso para ser exibido em sua totalidade, mas segue trecho suficiente para a compreensão:

```
Help on built-in module math:
```

```
NAME
```

```
    math
```

```
DESCRIPTION
```

```
    This module provides access to the mathematical functions
    defined by the C standard.
```

```
FUNCTIONS
```

```
    acos(x, /)
```

```
        Return the arc cosine (measured in radians) of x.
```

```
        The result is between 0 and pi.
```

```
    acosh(x, /)
```

```
        Return the inverse hyperbolic cosine of x.
```

```
    asin(x, /)
```

```
        Return the arc sine (measured in radians) of x.
```

```
        The result is between -pi/2 and pi/2.
```

```
    asinh(x, /)
```

```
        Return the inverse hyperbolic sine of x.
```

O comando `help()` acessa a documentação da biblioteca e oferece a descrição do módulo, exibindo as suas funções e explicando o que fazem.

CAPÍTULO 6 – ESTRUTURAS DE DECISÃO

6.1 Definição e Utilidade

Como vimos até agora em todos os nossos exemplos e programas, o interpretador do Python lê o código de cima para baixo em ordem sequencial. Blocos de código que estão mais acima são lidos antes do que aqueles que estão mais abaixo.

Isso é intuitivo e natural, pois também é assim que lemos textos. No entanto, você já se deparou com algum daqueles livros interativos que mandam o leitor tomar uma decisão e, dependendo do que escolherem, pular para determinada página? Imagine o seguinte trecho de um livro:

“Você está há horas caminhando perdido pelo deserto e sem água, quando, de repente, encontrou uma lâmpada parcialmente enterrada na areia. Deseja pegar esse item?

- Caso tenha pegado a lâmpada, pule para a página 15.
- Caso tenha optado por seguir em frente sem ela, continue lendo na página atual.”

As estruturas de decisão servem para fazer exatamente isso com o código. Impondo condições, podemos manipular a ordem de execução dos blocos de código, decidindo não somente o que será e o que não será executado, mas também quando será executado.

Essas estruturas, então, impõem condições que devem ser cumpridas para que determinado trecho do código seja executado. Tomando um exemplo simples: um professor usa o sistema da faculdade para calcular a média das notas das provas dos seus alunos. Os alunos que conseguiram média maior ou igual a 7 passaram de ano, enquanto os que não conseguiram atingir a média reprovaram. O sistema, com base nisso, enviará aos alunos mensagem confirmando a aprovação ou reprovação.

Esse é um típico exemplo de uso de estrutura de decisão. O código está pronto para lidar com ambos os casos: aprovação ou reprovação. O que irá decidir qual mensagem será enviada é o cumprimento (ou não) da condição imposta (média maior ou igual a sete).

Nesse exemplo, não importa o que o programador escreveu primeiro no código. O programa somente dará continuidade ao bloco de código que satisfaz a condição. Para visualizar melhor:

CONDIÇÃO: Média menor que 7

RESULTADO: Informar reprovação ao aluno

CONDIÇÃO: Média maior ou igual a sete

RESULTADO: Informar aprovação ao aluno

Observe que se a média foi maior ou igual a sete, o programa irá ignorar a execução do bloco de código que determinava “informar reprovação ao aluno”, mesmo esse bloco tendo sido escrito antes do resto do código que dizia o que fazer em caso de aprovação. Conseguimos manipular a ordem de execução das linhas de código do programa!

6.2 Estrutura if-else

A estrutura básica de decisão do Python consiste no if-else (se-senão). Nela passamos uma condição para o programa usando a palavra reservada “if” e posteriormente passamos a instrução que deve ser seguida caso a condição seja atendida. Depois, usamos a palavra reservada “else” e passamos outra instrução a ser seguida caso a condição não seja cumprida.

A sintaxe dessa estrutura é a seguinte:

```
if CONDIÇÃO:
```

```
    faça alguma coisa
```

```
else:
```

```
    faça outra coisa
```

É essencial que se use o sinal de dois pontos (:) para que o Python entenda que estamos criando um bloco condicional. Outro aspecto essencial é o espaço para o comando “faça alguma coisa”. Perceba que ele não está alinhado com o if nem com o else, mas sim separado por um “tab” ou quatro espaços em branco.

Essa é a forma de escrever o código para que a linguagem entenda que o programador criou um bloco de código usando a estrutura if-else. São essas palavras reservadas e esse espaço que indicam para o interpretador quando o bloco começa e quando ele termina.

O nome desses espaços é IDENTIFICAÇÃO. Então, quando falarmos em indentar, já sabemos que temos de dar um “tab” (ou quatro espaços em branco) antes de escrever o código que desejamos.

Vamos criar um programa simples para facilitar a compreensão. O programa irá receber um número inteiro como input do usuário e irá exibir mensagem indicando se o número digitado é par ou ímpar.

Para saber se o número é par ou ímpar, basta usarmos o resto da divisão por 2. Se o número pode ser dividido por 2 ele é par. Se nessa divisão o resto foi 1, ele é ímpar. O operador lógico que faz isso é o módulo (%) que estudamos no item 3.1.4. Como um

número somente pode ser par ou ímpar, não existindo meio termo, somente precisamos saber se ele é par, pois, caso não o seja, automaticamente sabemos que ele é ímpar.

Resumindo:

- a) o programa vai receber um número inteiro como input do usuário;
- b) o programa deve averiguar se esse número é par ou ímpar;
- c) se for par, o programa exibirá mensagem de que o número é par;
- d) senão, o programa exibirá mensagem de que o número é ímpar

Vamos escrever o código!

```
if_else.py x
1  numero = int(input("Digite um número inteiro: "))
2
3  # aqui estou fora do bloco if-else
4  if numero % 2 == 0:
5      print("O número digitado é par!")
6  else:
7      print("O número digitado é ímpar!")
8
9  # aqui estou fora do bloco if-else
10
```

Resultado:

```
Digite um número inteiro: 5
O número digitado é ímpar!

Process finished with exit code 0
```

Como o usuário digitou o número 5, o programa constatou que $5 \% 2$ (o resto da divisão entre 5 e 2) não é 0, a condição não foi cumprida. Como a condição não foi cumprida, o comando para printar “O número digitado é par!” não foi executado e o programa passou para a análise do nosso “senão” (else).

Como o comando para caso a condição não fosse cumprida era printar “O número digitado é ímpar!” e a condição realmente não foi cumprida, esse foi o comando executado.

Inserimos comentários indicando onde começa e onde termina o bloco da estrutura if-else para facilitar a compreensão. Como usamos as palavras reservadas seguidas de : e com o resto do código indentado, o Python sabe o início e o fim do bloco. Se não fizermos dessa maneira, teremos um erro e o programa irá quebrar!

6.2.1 Verificando Várias Condições Independentes

A estrutura if-else permite que verifiquemos mais de uma condição. No exemplo anterior, somente queríamos saber se o número era par ou ímpar, então somente essa condição deveria ser verificada.

Porém, não é obrigatório que o bloco condicional somente trate de uma condição. Podemos usar o “if” quantas vezes forem necessárias.

O exemplo clássico utilizado para demonstrar isso é o programa que calcula a média do aluno e exibe mensagens diferentes de acordo com a média calculada. Vamos explicar como seria esse programa.

O programa pede dois números decimais (float) como input e calcula a média aritmética para saber se o aluno foi aprovado ou reprovado. Caso a média seja menor do que 5, o programa exibirá a mensagem de reprovação. Se for superior a 5, o programa exibirá a mensagem de aprovação. No entanto, se o aluno passou com média 10, além da mensagem de aprovação, o programa também exibirá mensagem parabenizando o aluno por ter conseguido a nota máxima.

Para escrevermos o código:

- a) receber dois números decimais do usuário e armazená-los em variáveis;
- b) calcular a média aritmética e armazená-la numa variável;
- c) prever as condições para as exibições de cada tipo de mensagem.

Escrevendo o código:

```
1 nota_1 = float(input("Digite a nota da primeira prova: "))
2 nota_2 = float(input("Digite a nota da segunda prova: "))
3
4 media = (nota_1 + nota_2) / 2 # colocamos as notas entre () seguindo as regras da ordem das operações matemáticas
5
6 if media < 5:
7     print(f"A média final foi {media}. O aluno foi reprovado.")
8 if media >= 5:
9     print(f"A média final foi {media}. O aluno foi aprovado.")
10 if media == 10:
11     print("Parabéns! Aprovado com nota máxima!")
```

Resultado considerando as notas 7,5 e 8:

```
Digite a nota da primeira prova: 7.5
Digite a nota da segunda prova: 8
A média final foi 7.75. O aluno foi aprovado.
```

O Python analisou todas as condições e chegou à conclusão de que somente a segunda (média ≥ 5) foi satisfeita e por isso somente imprimiu a mensagem de aprovação. Vamos ver o que acontece com o programa se o aluno conseguir a média 10:

```
Digite a nota da primeira prova: 10
Digite a nota da segunda prova: 10
A média final foi 10.0. O aluno foi aprovado.
Parabéns! Aprovado com nota máxima!
```

Quando a média foi 10, o Python entendeu que tanto a segunda quanto a terceira condições foram atendidas. A média foi maior que 5 e, ao mesmo tempo, foi igual a 10. Como essas condições foram analisadas todas de forma independente, o programa executou os blocos de códigos que satisfizeram as condições.

Por isso, além da mensagem de aprovação, também recebemos a mensagem especial de parabenização pelo 10. Podemos estabelecer quantas condições independentes acharmos necessário para elaborar o nosso programa, não precisam ser somente três como usamos nesse exemplo das médias.

6.2.2 Mais Exemplos de Condições

É muito importante estabelecermos condições claras e precisas, sem ambiguidade, para que o programa consiga analisá-las sem problemas. Para isso, usamos os operadores relacionais e de igualdade para montar as condições que são necessárias para a execução dos nossos programas.

Vamos dar mais exemplos de formas de se escrever condições.

a) Usando AND:

Como já estudamos, quando usamos and o resultado somente será verdadeiro (True) se todas as expressões forem verdadeiras. Exemplo:

```
1 num = int(input("Digite um número inteiro: "))
2
3 if num % 2 == 0 and num % 3 == 0:
4     print("O número é múltiplo de 2 e de 3!")
5 else:
6     print("Ao menos uma das condições é falsa")
```

Resultado digitando 6 (é múltiplo de 2 e de 3):

```
Digite um número inteiro: 6
O número é múltiplo de 2 e de 3!
Process finished with exit code 0
```

Resultado digitando 4 (só é múltiplo de 2):

```
Digite um número inteiro: 4
Ao menos uma das condições é falsa
Process finished with exit code 0
```

A condição somente é satisfeita quando todas as expressões são verdadeiras (True). No exemplo usamos duas expressões, mas podemos usar quantas nós quisermos.

ATENÇÃO: é obrigatório deixar explícita a condição em cada expressão. Não podemos fazer algo direto do tipo IF NUM % 2 == 0 AND % 3 == 0. Se fizermos assim o programa irá quebrar ou o Python entenderá a expressão de forma errada e não a analisará corretamente. O computador não pensa e não consegue interpretar texto ou resolver ambiguidades como os humanos!

```

1 # ERRADO - isso vai gerar erro!
2 if num % 2 == 0 and % 3 == 0:

```

b) Usando OR:

Quando usamos or o resultado somente será verdadeiro (True) se ao menos uma das expressões forem verdadeiras. Exemplo:

```

1 num = int(input("Digite um número inteiro: "))
2
3 if num % 2 == 0 or num % 3 == 0:
4     print("O número é múltiplo de 2 ou é múltiplo de 3! É possível que seja múltiplo de ambos ao mesmo tempo!")
5 else:
6     print("Ambas as condições são falsas.")

```

Resultado digitando 10 (é múltiplo de 2):

```

Digite um número inteiro: 10
O número é múltiplo de 2 ou é múltiplo de 3! É possível que seja múltiplo de ambos ao mesmo tempo!
Process finished with exit code 0

```

Resultado digitando 5 (não é múltiplo de 2 e nem de 3):

```

Digite um número inteiro: 5
Ambas as condições são falsas.
Process finished with exit code 0

```

c) Usando intervalos:

Vamos pensar que determinado programa precisa executar alguma ação caso uma variável “num” tenha valor maior que 10, mas menor que 50. Podemos escrever a condição dessa maneira:

```

if num > 10 and num < 50:

```

Entretanto, o mais usual é escrever um intervalo. Sabemos que “num” deve estar em um intervalo entre 10 e 50 para que a condição seja cumprida. Basta escrevermos a condição usando a notação matemática de intervalo:

```

if 10 < num < 50:

```

Ambas as formas estão corretas, mas sempre que tivermos um intervalo devemos usar a segunda, pois é considerada melhor prática de programação.

6.3 Estrutura if – elif – else

Essa estrutura parte do mesmo princípio da estrutura condicional básica de if – else, mas adiciona um outro aspecto: o elif. Essa palavra reservada representa a combinação de else com if. Seria algo como “senão se” em português.

Assim como o if – else, essa estrutura também analisa a ocorrência de condições e sua sintaxe é idêntica, com a única diferença consistindo no uso da palavra reservada elif.

Enquanto na estrutura básica temos apenas condicionais independentes usando if e depois podemos finalizar com else para tratar casos nos quais nenhuma condição seja atendida, na estrutura if – elif – else começamos com um if para a primeira condição e usamos elif para as demais, finalizando, também, com o else.

As regras de sintaxe e indentação permanecem as mesmas:

```
if CONDIÇÃO:
```

```
    faça alguma coisa
```

```
elif CONDIÇÃO:
```

```
    faça outra coisa
```

```
else:
```

```
    faça outra coisa ainda
```

Essa estrutura é extremamente utilizada e representa condições exclusivas. Diferente da estrutura if – else, assim que uma condição for satisfeita, o Python irá ignorar as demais e nem irá se preocupar em averiguar se alguma outra condição também foi satisfeita.

Nessa estrutura somente um bloco de código será executado. Vamos utilizar o mesmo programa que calculava a média do aluno, com algumas modificações, para demonstrar o funcionamento dessa estrutura.

O que vamos alterar:

- a) O programa agora irá exibir mensagem de erro caso a média calculada seja inferior a 0 ou superior a 10 (significa que o usuário do programa inseriu números inválidos para as notas).
- b) Teremos mais condições a serem analisadas.
- c) **ATENÇÃO: O programa será criado com um pequeno “erro” para demonstrar que se uma condição é atendida, as demais sequer são consideradas.**

```
1  nota_1 = float(input("Digite a nota da primeira prova: "))
2  nota_2 = float(input("Digite a nota da segunda prova: "))
3
4  media = (nota_1 + nota_2) / 2
5
6  if 0 <= media < 3:
7      print(f"A média final foi {media}. O aluno foi reprovado.")
8  elif 3 <= media < 5:
9      print(f"A média final foi {media}. O aluno está de recuperação.")
10 elif 5 <= media < 7:
11     print(f"A média final foi {media}. O aluno foi aprovado.")
12 elif 7 <= media <= 10:
13     print(f"A média final foi {media}. O aluno foi aprovado com boa nota.")
14 elif media == 10:
15     print("Parabéns! Aprovado com nota máxima!")
16 else:
17     print("Média inválida.")
```

Esse programa irá rodar de forma muito similar ao programa original que usamos apenas if. O “erro” que mencionamos está nas linhas 14 e 15, pois a mensagem de nota máxima nunca será exibida. Vamos provar isso fazendo o aluno atingir média 10:

```
Digite a nota da primeira prova: 10
Digite a nota da segunda prova: 10
A média final foi 10.0. O aluno foi aprovado com boa nota.

Process finished with exit code 0
```

Por que isso acontece? Foi culpa do uso dos elif. Assim que o Python percebeu que a média estava entre 7 e 10 (linha 12), ele já executou o comando e parou de analisar as demais condicionais.

Caso mudemos o elif da linha 14 para if, o programa funcionará corretamente:

```
1  nota_1 = float(input("Digite a nota da primeira prova: "))
2  nota_2 = float(input("Digite a nota da segunda prova: "))
3
4  media = (nota_1 + nota_2) / 2
5
6  if 0 <= media < 3:
7      print(f"A média final foi {media}. O aluno foi reprovado.")
8  elif 3 <= media < 5:
9      print(f"A média final foi {media}. O aluno está de recuperação.")
10 elif 5 <= media < 7:
11     print(f"A média final foi {media}. O aluno foi aprovado.")
12 elif 7 <= media <= 10:
13     print(f"A média final foi {media}. O aluno foi aprovado com boa nota.")
14 if media == 10:
15     print("Parabéns! Aprovado com nota máxima!")
16 else:
17     print("Média inválida.")
```

Console:

```
Digite a nota da primeira prova: 10
Digite a nota da segunda prova: 10
A média final foi 10.0. O aluno foi aprovado com boa nota.
Parabéns! Aprovado com nota máxima!

Process finished with exit code 0
```

Agora conseguimos que a mensagem de aprovação também seja exibida.

6.4 Diferenças entre if e elif

Conforme explicamos nos itens anteriores, o uso de vários if indica ao Python que desejamos que as condições sejam independentes, ou seja, devem ser analisadas uma a uma.

Mesmo que uma condição tenha sido cumprida, o Python continuará analisando as demais, pois é possível que outras também tenham sido satisfeitas. Com o uso dessa estrutura, todos os blocos que tiverem as suas condições satisfeitas serão executados.

Por outro lado, conforme foi demonstrado pelo último programa, quando usamos `elif` estamos dizendo ao Python para executar o bloco de código da primeira condição que for satisfeita. Ainda que haja a possibilidade de outras condicionais também terem sido cumpridas, o Python irá ignorá-las e somente executará a primeira que ele encontrar como satisfeita.

Quando usar vários `if` de forma independente? Quando precisamos que cada condição seja analisada de forma individual e independente. O programa ficará mais lento, pois todas as condições deverão ser checadas individualmente.

Quando usar `elif`? Quando somente o cumprimento de uma das condições já basta. O programa rodará mais rápido, pois assim que uma condição foi satisfeita, todas as demais serão ignoradas.

6.5 Estrutura Condicional Aninhada

Esse tipo de estrutura tem esse nome porque é uma condição dentro de outra condição. Também usamos estruturas `if – else` ou `if – elif -else` normalmente, sendo que a única coisa que muda é que temos uma condição principal a ser cumprida, sendo que as condições aninhadas dependem dessa principal para serem analisadas.

Ou seja, essas condições secundárias dependem da condição principal. Caso a condição principal não seja satisfeita, essas condições aninhadas serão ignoradas. Por outro lado, se a condição principal for cumprida, o programa passará a analisar as condições secundárias.

Um bom exemplo seria um programa que analisa se o homem está em idade para serviço militar. Primeiro descobrimos se o usuário é homem ou mulher. Se for mulher, o serviço militar não é obrigatório e o programa para nesse ponto. Porém, se for homem, há a condição secundária de ser maior de idade.

Essa condição de ser maior de idade somente é analisada se a condição principal (ser homem) tiver sido cumprida. É uma condição aninhada.

A sintaxe é muito simples: da mesma forma que usamos indentação para indicar o bloco de código, usaremos mais uma indentação para o bloco aninhado. Visualmente:

if CONDIÇÃO PRINCIPAL:

 faça alguma coisa

 if CONDIÇÃO SECUNDÁRIA:

 faça alguma coisa

 else:

 faça outra coisa

else:

 faça outra coisa

A indentação indica para o Python o começo e o fim tanto do bloco principal quanto do bloco aninhado. Ela é obrigatória e deve ser feita com cuidado.

Vamos criar o nosso programa. Para isso precisaremos:

- a) Perguntar ao usuário se ele é homem;
- b) Se for mulher o serviço militar não é obrigatório;

- c) Se for homem, devemos perguntar se ele tem 18 anos ou mais (condição aninhada);
- d) O serviço militar somente é obrigatório se ele for maior de idade

Escrevendo o código:

```
1  sexo = input("O seu sexo é masculino? [s/n] ")
2
3  if sexo == "s":
4      idade = int(input("Informe a sua idade: "))
5      if idade >= 18:
6          print("Você deve realizar o serviço militar obrigatório")
7      else:
8          print("Você é menor de idade e não precisa se alistar.")
9  else:
10     print("Você está dispensada do serviço militar obrigatório")
11
```

É algo bem simples. Nada mais é do que uma condicional dentro de outra condicional. Podemos usar tanto estruturas com if quanto com elif.

Vamos ver o resultado do programa quando é usado por uma mulher:

```
O seu sexo é masculino? [s/n] n
Você está dispensada do serviço militar obrigatório

Process finished with exit code 0
```

Como a condição principal (sexo masculino) não foi cumprida, o programa foi direto para o else mais externo (no final, linha 9).

Agora vamos ver um homem maior de idade:

```
O seu sexo é masculino? [s/n] s
Informe a sua idade: 19
Você deve realizar o serviço militar obrigatório

Process finished with exit code 0
```

Aqui a condição principal foi cumprida. Como o homem tinha 19 anos (maior de 18), a condição secundária também foi cumprida e a mensagem de alistamento foi exibida.

Vamos ver o homem menor de idade:

```
O seu sexo é masculino? [s/n] s
Informe a sua idade: 14
Você é menor de idade e não precisa se alistar.

Process finished with exit code 0
```

Por ser homem, o programa passou a analisar a condição aninhada. Porém, ela não foi cumprida (o usuário era menor de idade) e por isso o programa executou o que estava previsto no else interno da condição aninhada (else da linha 7).

O funcionamento dessa estrutura é idêntico ao funcionamento daquelas que estudamos. Somente precisamos ter cuidado com a indentação correta para não quebrarmos o programa.

CAPÍTULO 7 – Tratamento Básico de Dados Recebidos do Usuário

7.1 Dados Incorretos

Como já vimos em diversos exemplos, é possível que o usuário não siga as instruções do prompt de input corretamente e alimente o programa com dados que irão provocar erros, com a quebra do programa, ou que farão o programa não funcionar corretamente.

Existem formas de preparar o programa para identificar que os dados inseridos irão causar problema e, se for o caso, informar isso ao usuário e pedir novamente que ele insira os dados de forma adequada.

Esse tipo de tratamento geralmente é feito por meio de exceções, o que é um assunto um pouco mais complexo e não será abordado nesta parte tutorial (estudaremos em outra parte). O programador, no momento, somente deve saber que existe uma forma muito eficiente de lidar com eventuais problemas de input.

O que vamos abordar nesse capítulo são formas básicas de tratar os erros mais comuns que os usuários costumam cometer ao fornecerem o seu input ao programa. Vamos aprender como lidar com espaços em branco digitados por engano, como fazer o programa aceitar que o usuário use letras maiúsculas ou minúsculas sem diferenciação e como exibir mensagens criadas por nós em caso de erro (em vez daquelas mensagens automáticas de erro vermelhas que já vimos acontecer).

7.2 Lidando com Espaços – strip()

Um dos erros mais comuns que o usuário pode cometer é passar espaços indesejados antes e/ou depois do input que desejava. Por exemplo: o programa pede para digitar um número inteiro e o usuário digita 9, mas com um espaço em branco antes porque o dedo esbarrou na tecla de espaço e ele não percebeu.

Esse tipo de input pode quebrar o programa. Para evitar que esse problema ocorra, podemos usar a função `strip()`. Ela funciona da seguinte maneira: remove todos os espaços em brancos que estejam antes ou depois do que o usuário digitar. Se não houver espaços, nada acontece.

Para fazer uso dessa função, basta usar a string seguida de um ponto final (.) e o `strip()` – `string.strip()`. Como sabemos que o `input()` nos dá uma string (já provamos isso no item 3.4), seguiremos essa sintaxe: `variável = input("prompt").strip()`.

Vamos dar um exemplo. Iremos criar duas variáveis diferentes que pedirão o input do usuário. Iremos enviar como input a frase “ Hello, world!”. O primeiro input será normal, já o segundo usará a função `strip()`. Em seguida vamos printar ambas as variáveis para vermos o `strip()` em ação.

```
tratamento_basico.py x
1 texto_1 = input("Digite qualquer coisa com espaços antes: ")
2 texto_2 = input("Digite a mesma coisa com espaços antes: ").strip()
3
4 print(texto_1)
5 print(texto_2)
6
```

Executando o programa e inserindo o texto:

```
Digite qualquer coisa com espaços antes: Hello, world!
Digite a mesma coisa com espaços antes: Hello, world!
Hello, world!
Hello, world!

Process finished with exit code 0
```

Veja que o “Hello, world!” com `strip()` teve os seus espaços anteriores ao início da frase removidos. Conforme já explicamos, se não houver espaços antes e/ou depois do input a função não vai fazer nada, de forma que não há risco nenhum de atrapalhar o programa.

7.3 Lidando com Maiúsculas e Minúsculas – upper() e lower()

Outro problema que pode acontecer vem da diferenciação de letras maiúsculas e minúsculas. Como o Python entende essa diferença, é possível que o usuário não se atente para essa questão e insira um dado que não será reconhecido pelo programa.

As funções upper() e lower() vão garantir que tudo o que o usuário digitar será passado, respectivamente, para letras maiúsculas ou minúsculas. A sintaxe dessas funções é exatamente igual à da strip(): string.upper() e string.lower().

O programa do alistamento militar que fizemos no capítulo anterior é muito bom para mostrar o funcionamento. Relembrando:

```
1  sexo = input("O seu sexo é masculino? [s/n] ")
2
3  if sexo == "s":
4      idade = int(input("Informe a sua idade: "))
5      if idade >= 18:
6          print("Você deve realizar o serviço militar obrigatório")
7      else:
8          print("Você é menor de idade e não precisa se alistar.")
9  else:
10     print("Você está dispensada do serviço militar obrigatório")
11
```

O que aconteceria se um homem passasse como input um “S” maiúsculo em vez de minúsculo? O programa somente previu condição do usuário responder como “s” para afirmar que era homem. Ele vai entender que “S” também significa que o usuário quer responder que é homem?

```
O seu sexo é masculino? [s/n] S
Você está dispensada do serviço militar obrigatório

Process finished with exit code 0
```

Resposta: não! Como passamos “S” e a condição da linha 3 apenas verifica se a resposta foi “s”, o programa não entendeu que o usuário quis dizer que era do sexo masculino.

Vamos resolver isso usando `lower()`, pois assim mesmo que o usuário digite em caixa alta, o conteúdo da variável “sexo” terá letra minúscula:

```
1  sexo = input("O seu sexo é masculino? [s/n] ").lower()
2
3  if sexo == "s":
4      idade = int(input("Informe a sua idade: "))
5      if idade >= 18:
6          print("Você deve realizar o serviço militar obrigatório")
7      else:
8          print("Você é menor de idade e não precisa se alistar.")
9  else:
10     print("Você está dispensada do serviço militar obrigatório")
```

Executando com input em caixa alta:

```
O seu sexo é masculino? [s/n] S
Informe a sua idade: 18
Você deve realizar o serviço militar obrigatório

Process finished with exit code 0
```

Agora mesmo tendo digitado “S”, a variável `sexo` recebeu “s” em letra minúscula graças à função `lower()`.

ATENÇÃO: É possível combinar essas funções uma em seguida da outra. Podemos usar `string.strip().lower()`, por exemplo, para garantirmos que não teremos espaços antes e depois do input e que o texto desse input será com letras minúsculas:

```
1  sexo = input("O seu sexo é masculino? [s/n] ").strip().lower()
```

Isso funciona!

7.4 Enviando Nossas Mensagens de Erro

Toda vez que o programa quebrar, receberemos aquela mensagem de erro vermelha do próprio Python indicando qual foi o erro e onde ele aconteceu. Lembrando:

```
Traceback (most recent call last): @ Explain with AI
  File "C:\Users\rafae\AppData\Roaming\JetBrains\PyCharmCE2024.3\light-edit\main.py", line 7, in <module>
    resultado_divisao = num1 / num2
                        ~~~~~^~~~~~
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Usando estruturas condicionais, podemos prever possíveis inputs indesejáveis por parte do usuário e determinar que o programa tome ações que julgarmos necessárias, como, por exemplo, exibir uma mensagem no console criada por nós usando o comando `print()`.

Para podermos fazer isso bem, vamos, primeiro, aprender sobre os operadores de pertinência, que mencionamos no item 3.1.7, mas ainda não estudamos.

7.4.1 Operadores de Pertinência

Esses operadores servem para averiguar se determinado membro pertence a uma sequência ou estrutura de dados. Podem ser usados com strings (já aprendemos), listas, tuplas, dicionários e conjuntos. Ainda não estudamos esses tópicos, mas não se preocupe. Apenas saiba que existem essas estruturas e que elas serão apresentadas em outra parte desse tutorial.

Por enquanto sabemos que vamos usar esses operadores com strings. Vamos supor a palavra “Python” e queremos saber se temos a letra “j” nesse texto. Os operadores de pertinência servem exatamente para essa situação: saber se algo está contido (ou não) nesse texto.

São eles:

- a) `in`
- b) `not in`

Vamos ver exemplos:

```
1 print("j" in "Python") # False, pois não tem letra j na palavra Python
2 print("a" in "casa") # True, pois tem a letra a na palavra casa
3 print("j" not in "Python") # True, pois não tem letra j na palavra Python
4 print("b" not in "banana") # False, pois tem a letra b na palavra banana
5
```

Resultado:

```
False
True
True
False

Process finished with exit code 0
```

Esses simples operadores irão nos ajudar nesse objetivo de enviarmos nossas próprias mensagens quando o usuário não seguir as instruções corretamente.

ATENÇÃO: Letras maiúsculas e minúsculas são diferenciadas. Tentar achar “b” em “Banana” retornará False.

7.4.2 Realizando O Tratamento

Agora já temos todo o conhecimento necessário para continuar. Vamos melhorar o nosso programa de alistamento militar, prevendo algumas situações e problemas que podem acontecer.

Em primeiro lugar, vamos usar `strip()` e `lower()` para garantir que o input do usuário virá sem espaços e em letra minúscula. Em seguida, vamos garantir que o usuário esteja obrigado a responder a pergunta com “s” ou “n”. Caso ele digite algo diferente disso, o programa irá parar e exibir a mensagem de que os dados enviados foram inválidos.

Para isso, vamos usar uma condicional que verifica se o valor digitado está presente em uma palavra que vamos criar – “sn”. Caso o que foi digitado seja “s” ou “n”

(ambos os caracteres estão presentes na palavra “sn”), o programa irá continuar rodando. Caso contrário, a mensagem de erro será exibida.

Por fim, vamos impedir que o usuário insira valores negativos para a idade. Caso ele faça isso, o programa também irá enviar uma mensagem de erro explicando que os dados são inválidos.

Aplicando as melhorias:

```
1  sexo = input("O seu sexo é masculino? [s/n] ").strip().lower()
2
3  if sexo not in "sn":
4      print("Dados inválidos. Por favor, reinicie o programa.")
5  elif sexo == "s":
6      idade = int(input("Informe a sua idade: "))
7      if idade < 0:
8          print("Dados inválidos. Por favor, reinicie o programa.")
9      elif idade >= 18:
10         print("Você deve realizar o serviço militar obrigatório")
11     else:
12         print("Você é menor de idade e não precisa se alistar.")
13 else:
14     print("Você está dispensada do serviço militar obrigatório")
15
```

Resultado caso o usuário tente mandar um número como input para a pergunta do sexo:

```
O seu sexo é masculino? [s/n] 3
Dados inválidos. Por favor, reinicie o programa.

Process finished with exit code 0
```

Percebam que o programa exibiu a mensagem que escrevemos, conforme esperávamos.

Resultado caso o usuário responda a pergunta com “s”, mas tente passar uma idade negativa:


```
O seu sexo é masculino? [s/n] s
Informe a sua idade: -2
Dados inválidos. Por favor, reinicie o programa.

Process finished with exit code 0
```

Mais uma vez, o nosso código previu essa possibilidade e sabia o que fazer caso isso acontecesse. Nossa mensagem foi exibida.

7.5 Considerações Finais

Essa forma de tratamento de dados é básica e inicial. Existem formas mais complexas e melhores para tratar os possíveis erros (em Python chamados de exceções). Entretanto, esse estudo ficará para outra parte desse tutorial, por ser mais complicado e ser conveniente que tenhamos mais conhecimentos antes de falarmos sobre isso.