3

TEXTO BASE

LINGUAGEM DE PROGRAMAÇÃO

Faculdade IMPACTA

Texto base

3

Expressões, operadores de atribuição, funções integradas e entrada/saída de dados

Prof. Me. Lucio Nunes de Lira
Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

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

3.1. Motivação

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

💎 VOCÊ CONHECE?

Ada Lovelace, matemática, escritora e reconhecida como a primeira programadora do mundo! Escreveu um algoritmo que poderia ser processado na máquina analítica de Charles Babbage. A linguagem de programação Ada foi batizada em sua homenagem, assim como diversos eventos e premiações científicas.
Fonte da imagem: https://www.biography.com/scholar/ada-lovelace

3.2. Expressões

Uma expressão é uma combinação de operandos e operadores, resultando em um valor, sendo que os operandos podem ser, por exemplo, constantes e variáveis. Para que uma expressão esteja correta, essa combinação precisa ser válida. Um único operando já é considerado uma expressão válida. Também podemos usar parênteses, como na matemática. Teste na *Shell* a Codificação 3.1.

```
>>> 3.1415   # expressão válida, resulta em 3.1415
>>> (2 + 5)  # expressão válida, resulta em 7
>>> 3.0 * 2  # expressão válida, resulta em 6.0
>>> 10 / 0   # expressão válida, porém resulta em erro de
               aritmética (erro de divisão por zero)
>>> +123     # expressão válida, resulta em 123
>>> 'oi'     # expressão válida, resulta em 'oi'
>>> 123+     # expressão inválida, falta um operando à direita
>>> 2 * 3)   # expressão inválida, parênteses desbalanceados
```

Codificação 3.1: Exemplos de expressões válidas e inválidas.

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

3.3. Operadores de atribuição

Conhecemos o operador de *atribuição simples* de Python (`=`), e entendemos que seu papel é atribuir valores de expressões às variáveis, como na Codificação 3.2.

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

Codificação 3.2: Exemplo de atribuição de valor à variável.

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

Em Python, há outros operadores de atribuição, chamados de operadores de *atribuição composta*, ou *atribuição aumentada*, usados para tornar algumas instruções mais concisas e, dependendo da maturidade do programador, mais legíveis. Veja parte desses operadores na Tabela 3.1. Também é possível substituir instruções com atribuição composta por atribuição simples combinada com outro operador.

Podemos dizer que estes operadores são um atalho de sintaxe para quando precisamos acessar o valor de uma variável, realizar uma operação com ele, e atribuir o resultado da operação à mesma variável. Teste a Codificação 3.3 e verifique se compreendeu a razão dos valores atribuídos a variável `m` a cada nova instrução,

lembre-se que para acessar o valor da variável `m`, basta inserir na *Shell* `>>> m` e pressionar `[ENTER]`.

Tabela 3.1: Relação parcial dos operadores de atribuição composta do Python.

Operador	Descrição	Exemplos
`=`	Atribui o resultado da expressão à variável.	`x = expressão`
`+=`	Atribui o resultado da adição à variável.	`x += expressão` `x = x + (expressão)`
`-=`	Atribui o resultado da subtração à variável.	`x -= expressão` `x = x - (expressão)`
`*=`	Atribui o resultado da multiplicação à variável.	`x *= expressão` `x = x * (expressão)`
`/=`	Atribui o quociente da divisão real à variável.	`x /= expressão` `x = x / (expressão)`
`//=`	Atribui o quociente da divisão inteira à variável.	`x //= expressão` `x = x // (expressão)`
`%=`	Atribui o resto da divisão inteira à variável.	`x %= expressão` `x = x % (expressão)`
`**=`	Atribui o resultado da exponenciação à variável.	`x **= expressão` `x = x ** (expressão)`

```
>>> m = 2      # m recebe 2
>>> m **= 5    # m recebe 32
>>> m //= 4    # m recebe 8
>>> m += 1     # m recebe 9
>>> m /= 2     # m recebe 4.5
>>> m %= 2     # m recebe 0.5
>>> m -= 0.2   # m recebe 0.3
>>> m *= 5     # m recebe 1.5
```

Codificação 3.3: Uma mesma variável recebendo diversas atribuições.

VOCÊ SABIA?

Esses atalhos de sintaxe são comumente chamados de açúcar sintático, do inglês *syntactic sugar,* pois são formas de expressar algo em uma linguagem de programação de modo mais simples, claro ou conciso, mesmo que já exista outra forma equivalente. O intuito é facilitar a leitura e escrita de código. Portanto, são recursos que poderiam ser removidos e mesmo assim a linguagem não perderia nenhuma funcionalidade, embora poderia perder alguma conveniência.

Teste a Codificação 3.4 na *Shell* e verifique se compreendeu a razão dos valores atribuídos às variáveis.

```
>>> a = 10           # a recebe 10
>>> b = 10           # b recebe 10
>>> c = 10           # c recebe 10
>>> a *= 2 + 3       # a recebe 50
>>> b = b * 2 + 3    # b recebe 23
>>> c = c * (2 + 3)  # c recebe 50
```

Codificação 3.4: Atribuição composta e atribuição simples equivalente.

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

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

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

```
>>> x = 10           # x recebe 10
>>> y = 10           # y recebe 10
>>> z = 10           # z recebe 10
>>> x += 2 * 3       # x recebe 16
>>> y = y + 2 * 3    # y recebe 16
>>> z = z + (2 * 3)  # z recebe 16
```

Codificação 3.5: Atribuição composta e atribuição simples equivalente.

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

3.4. Precedência e associatividade de operadores

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

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

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

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

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

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

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

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

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

(II) porque representam algo “especial”, uma interpretação diferenciada na linguagem.

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

Tabela 3.2: Precedência e associatividade dos operadores vistos até agora.

Operador	Descrição	Associatividade
`**`	Exponenciação	À direita
`+`, `-` (unários)	Identidade e negação (inversão de sinal)	À esquerda
`*`, `/`, `//`, `%`	Multiplicação, divisão real, divisão inteira e resto da divisão	
`+`, `-` (binários)	Adição e subtração	
`=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`	Atribuição (simples e composta)	Não associativos

3.5. Avaliação de expressões

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

```
>>> a = 5
>>> b = 4
>>> c = 9
>>> d = 7
>>> e = 1
>>> f = 2
>>> s = 10
>>> s += a + b ** (c - d) / e * f
>>> s   # qual o valor de s?
```

Codificação 3.6: 1º expressão que será analisada e avaliada como em Python.

1) O par de parênteses prioriza a expressão interna, dando a ela maior precedência, porém, como a operação contém variáveis, primeiro devemos substituí-las por seus respectivos valores. Após ser avaliado, o par de parênteses é descartado:
   `s += a + b ** (9 - 7) / e * f` ⇨ `s += a + b ** 2 / e * f`
2) Precedência: `s += a + 4 ** 2 / e * f` ⇨ `s += a + 16 / e * f`
3) Precedência e associativ.: `s += a + 16 / 1 * f` ⇨ `s += a + 16.0 * f`
4) Precedência: `s += a + 16.0 * 2` ⇨ `s += a + 32.0`
5) Precedência: `s += 5 + 32.0` ⇨ `s += 37.0`
6) Único operador: `s += 37.0`
   ⇨ Atribuição composta, pode ser reescrita como: `s = s + (37.0)`
   ⇨ `s = 10 + (37.0)` ⇨ `s = 10 + 37.0` ⇨ `s = 47.0`

```
>>> t = 1 + 2 - 3 + 4 - 5
>>> t   # qual o valor de t?
```

Codificação 3.7: 2º expressão que será analisada e avaliada como em Python.

1) Associatividade: `t = 1 + 2 - 3 + 4 - 5` ⇨ `t = 3 - 3 + 4 - 5`
2) Associatividade: `t = 3 - 3 + 4 - 5` ⇨ `t = 0 + 4 - 5`
3) Associatividade: `t = 0 + 4 - 5` ⇨ `t = 4 - 5`
4) Associatividade: `t = 4 - 5` ⇨ `t = -1`
5) Único operador: `t = -1`

```
>>> u = 2 ** 1 ** 3
>>> u   # qual o valor de u?
```

Codificação 3.8: 3º expressão que será analisada e avaliada como em Python.

1) Associatividade: `u = 2 ** 1 ** 3` ⇨ `u = 2 ** 1`
2) Associatividade: `u = 2 ** 1` ⇨ `u = 2`
3) Único operador: `u = 2`

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

```
>>> v = ((2 ** 1) ** 3)
>>> v   # qual o valor de v?
```

Codificação 3.9: 4º expressão que será analisada e avaliada como em Python.

1) Parênteses: `v = ((2 ** 1) ** 3)` ⇨ `v = (2 ** 3)`
2) Parênteses: `v = (2 ** 3)` ⇨ `v = 8`
3) Único operador: `v = 8`

```
>>> w = (5*8) + ((9//7) % 2)
>>> w   # qual o valor de w?
```

Codificação 3.10: 5º expressão que será analisada e avaliada como em Python.

1) Parênteses: `w = (5*8) + ((9//7) % 2)` ⇨ `w = 40 + ((9//7) % 2)`
2) Parênteses: `w = 40 + ((9//7) % 2)` ⇨ `w = 40 + (1 % 2)`
3) Parênteses: `w = 40 + (1 % 2)` ⇨ `w = 40 + 1`
4) Precedência: `w = 40 + 1` ⇨ `w = 41`
5) Único operador: `w = 41`

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

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

Codificação 3.11: 6º expressão que será analisada e avaliada como em Python.

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

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

Codificação 3.12: 7º expressão que será analisada e avaliada como em Python.

A Codificação 3.12 é inválida, pois em Python a atribuição não resulta em valor, logo não há o que atribuir à `c`. Note que o `=` entre parênteses está fora do encadeamento.

```
>>> x = 4
>>> y = 5
>>> x += y += 1 # quais os valores das variáveis?
```

Codificação 3.13: 8º expressão que será analisada e avaliada como em Python.

A Codificação 3.13 é inválida, pois operadores de atribuição composta são não associativos e quando encadeados geram erro.

3.6. Funções

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

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

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

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

3.6.1. Funções integradas

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

Para usar funções integradas, também conhecidas como *built-in functions*, é necessário *chamá-las*. Para chamar funções, escreve-se seu nome seguido por um par de parênteses. Ao chamar a função solicitamos sua execução, como na Codificação 3.14.

```
>>> print()  # chama a função print.
```

Codificação 3.14: Chamada à função `print`.

Além de seus nomes e parênteses, a maioria das funções exige um terceiro componente: *argumentos*. Argumentos são valores inseridos entre os parênteses da chamada e servem para *passar* dados à função. Os dados passados são necessários para que a função cumpra seu objetivo. Por exemplo, a função `abs` *recebe* como argumento um número e *retorna* o valor absoluto correspondente, conforme a Codificação 3.15.

```
>>> abs(-29)  # chama a função abs com o argumento -29.
29            # valor retornado pela função abs.
```

Codificação 3.15: Chamada à função `abs` com o argumento `-29`.

O *valor retornado* ou *valor devolvido* é, como a própria expressão indica, um valor que será dado como retorno pela função, ou seja, a resposta obtida ao chamar a

função. Esse valor é o resultado da *avaliação da função*, algo semelhante ao que ocorre quando avaliamos expressões. Podemos associar os argumentos como as entradas da função e o valor de retorno como a saída da função.

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

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

```
>>> x = abs(-29)  # atribui o valor de retorno de abs à x.
>>> x + 15
44
```

Codificação 3.16: Atribuição do valor de retorno de `abs` à uma variável.

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

```
>>> abs(-29) + 15  # soma 15 ao valor retornado por abs.
44
```

Codificação 3.17: Chamada à uma função usada como operando em uma expressão.

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

Existem funções que requerem a passagem de mais de um argumento, neste caso eles deverão ser passados na ordem correta e separados por vírgulas. Veja, por exemplo, a função `pow` que retorna como resposta a potência de uma base elevada a um expoente, semelhante ao operador de exponenciação. Vamos refletir. Para que `pow` possa realizar a exponenciação, quais os dados necessários? Exatamente! A base e o expoente que se pretende calcular! O primeiro argumento é a base e o segundo é o expoente, como vemos na Codificação 3.18.

```
>>> pow(2, 5)  # 2 é a base e 5 o expoente
32
```

Codificação 3.18: Chamada à uma função com dois argumentos.

Quem determina quais os argumentos e a ordem correta de passá-los à função é seu criador, que pode disponibilizar documentação, geralmente na internet. Também podemos usar o menu *Help* do IDLE, caso seja uma das funções que acompanham a instalação. A própria *Shell* e o editor podem ajudar, basta digitar o nome da função com a abertura de parênteses e aguardar alguns instantes, conforme Figura 3.1.

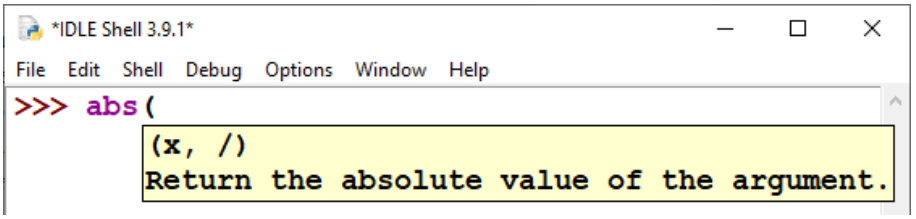

Figura 3.1: Resumo da documentação da função `abs`. Fonte: Elaborado pelo autor.

VAMOS LER!

O Python 3.9.1 possui 69 funções integradas, número que pode variar entre as versões. Veja a relação atualizada em: https://docs.python.org/pt-br/3/library/functions.html

3.7. Saída de dados

Você deve ter notado a função embutida `print` em algumas codificações, apesar de intuitiva por conta do efeito evidente em sua execução, aguardávamos a explicação inicial sobre funções para melhor compreensão. Esta função serve para *exibir* dados e será bastante usada durante a disciplina. Veja exemplos na Codificação 3.19.

```
>>> print('Hello World!')          # print com um argumento.
Hello World!
>>> print('2 + 2 =', 2+2)         # print com dois argumentos.
2 + 2 = 4
>>> idade = 29
>>> print('Tenho', idade, 'anos')  # print com três argumentos.
Tenho 29 anos
```

Codificação 3.19: Exemplos de uso da função `print`.

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

Lembre-se que códigos-fonte sem `print`, quando executados, não exibem nada. Para confirmar, insira a Codificação 3.19 no editor e execute-a com o atalho `[F5]`.

```
nome = 'Megan'
nome
idade = 34
idade
```

Codificação 3.20: Programa sem saída de dados.

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

```
nome = 'Megan'
print(nome)
idade = 34
print(idade)
```

Codificação 3.21: Programa com saída de dados.

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

Por padrão, `print` exibe os valores de seus argumentos e uma quebra de linha `'\n'`, fazendo com que o próximo `print` exiba seus dados na linha seguinte. Às vezes esse comportamento é indesejado, como na Codificação 3.22.

```
print('boa ')
print('noite ')
print('vizinhança')
```

Codificação 3.22: Programa em que as palavras deveriam estar na mesma linha.

Se quisermos que apenas o último `print` quebre a linha, basta acrescentar como último argumento da função um `end`, seguido de um sinal `=` com um valor à direita que será exibido no lugar da quebra de linha. Veja um exemplo na Codificação 3.23

```
print('boa ', end='')
print('noite ', end='')
print('vizinhança')
```

Codificação 3.23: Programa em que apenas o último `print` quebrará uma linha.

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

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

3.8. Entrada de dados

Para permitir que o usuário do programa forneça dados de entrada, usaremos a função embutida `input` que, quando executada, faz com que: (I) o programa aguarde um valor de entrada antes de prosseguir com as próximas instruções; (II) converte o valor lido para *string* e; (III) retorna-o como resposta. Há exemplo na Codificação 3.24.

```
>>> nome = input('Qual o seu nome? ')
Qual o seu nome? Megan
>>> print(nome)
Megan
```

Codificação 3.24: Exemplo de utilização da função `input`.

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

```
nome = input('Seu nome: ')
valor = input('Valor do empréstimo: ')
parcelas = input('Quantidade de parcelas: ')
print(nome, ', a dívida será de:', valor * parcelas)
```

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

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

As funções `int` e `float` recebem como argumento um número, ou uma *string* que represente adequadamente um número do tipo para qual se deseja converter, e devolvem como resposta o valor correspondente convertido para um número inteiro ou em ponto flutuante (real), respectivamente. Veja a correção da Codificação 3.25, agora com as devidas conversões, na Codificação 3.26.

```
nome = input('Seu nome: ')
valor = float(input('Valor do empréstimo: '))
parcelas = int(input('Quantidade de parcelas: '))
print(nome, ', a dívida será de:', valor * parcelas)
```

Codificação 3.26: Programa com conversões dos valores retornados por `input`.

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

1) Inicialmente `input` será executada, pois é a função mais interna:
`valor = float(input('Valor do empréstimo: '))`

2) Supondo que o usuário dê a entrada `1000.00`, a função `input` converterá esse valor para *string* e a retornará como resposta:
`valor = float('1000.00')`

3) Agora a função `float` converterá seu argumento, que neste caso é uma *string*, para o correspondente número real e retornará esse valor.
`valor = float('1000.00')` ⇨ `valor = 1000.0`

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

3.9. Tipos de erros comuns

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

1) *Erros de sintaxe:* ocorrem por falha na escrita das instruções da linguagem, violando regras e estruturas. Por exemplo, a instrução `print 'olá'`, sem parênteses, é inválida em Python 3, mas é válida em Python 2. Esquecer de completar toda abertura de parênteses com um parêntese de fechamento ou usar um operador binário com apenas um operando são outros exemplos. Geralmente são erros de simples resolução, pois o ambiente de programação costuma indicar instruções com erros de sintaxe ao executar o código;

2) *Erros em tempo de execução:* são erros que só aparecem quando o programa é executado, pois a sintaxe está correta. Também podem ser chamados de exceções, indicando que algo excepcional ocorreu, como uma violação do uso esperado do programa. Podem ocorrer, por exemplo, quando o usuário insere um valor inválido ou quando um recurso necessário para a execução do programa está ausente. O exemplo a seguir gera esse tipo de erro:

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

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

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

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

VOCÊ SABIA?

Existe uma empresa que mantém um indicador de popularidade sobre linguagens de programação e com atualizações mensais! O site é: https://www.tiobe.com/tiobe-index/

Em 2020, Python ganhou pela 4ª vez o prêmio "Linguagem de Programação do Ano", dado à linguagem com maior crescimento em popularidade no ano. No site consta o método usado para medir a popularidade.

Bibliografia e referências

PSF. **Expressions**. 2020. Disponível em: <https://docs.python.org/3/reference/expressions.html>. Acesso em: 21 jan. 2021.

STURTZ, J. Operators and Expressions in Python. **Real Python**, 2018. Disponível em: <https://realpython.com/python-operators-expressions/>. Acesso em: 21 jan. 2021.

STURTZ, J. Basic Input, Output, and String Formatting in Python. **Real Python**, 2019. Disponível em: <https://realpython.com/python-input-output/>. Acesso em: 21 jan. 2021.