FUNDAMENTOS DE PROGRAMAÇÃO

SUMÁRIO

AVALIANDO EXPRESSÕES

Uma expressão é uma combinação de valores, variáveis e operadores. Se você digitar uma expressão na linha de comando, o interpretador avalia e exibe o resultado:

```
1 + 1
```

```
2
```

Embora expressões contenham valores, variáveis e operadores, nem toda expressão contém todos estes elementos. Um valor por si só é considerado uma expressão, do mesmo modo que uma variável. Avaliar uma expressão não é exatamente a mesma coisa que imprimir um valor:

```
mensagem = "E aí, Doutor?"
```

```
>>> mensagem
```

```
'E aí, Doutor?'
```

```
>>> print (mensagem)
```

```
E aí, Doutor?
```

COMANDOS

Um comando é uma instrução que o interpretador Python pode executar. Vimos até agora dois tipos de comandos: de exibição (a chamada da função print) e de atribuição (=).

Quando você digita um comando na linha de comando, o Python o executa e mostra o resultado, se houver um. O resultado de um comando como a chamada da função print é a exibição de um valor. Comandos de atribuição não produzem um resultado visível. Um script normalmente contém uma sequência de comandos. Se houver mais de um comando, os resultados aparecerão um de cada vez, conforme cada comando seja executado.

Por exemplo, o “script”:

```
print (1)
```

```
x = 2
```

```
print (x)
```

produz a saída:

```
1
```

```
2
```

EXPRESSÃO E VARIÁVEL

Quando Python exibe o valor de uma expressão, usa o mesmo formato que você usaria para entrar com o valor. No caso de strings, isso significa que as aspas são incluídas. Mas o comando print imprime o valor da expressão, que, neste caso, é o conteúdo da string.

Num script, uma expressão sozinha é um comando válido, porém sem efeito. O script:

```
17

3.2

"Alô, Mundo!"

1 + 1
```

Não produz qualquer saída.

Quando Python exibe o valor de uma expressão, usa o mesmo formato que você usaria para entrar com o valor.

OPERADORES E OPERANDOS

Operadores são símbolos especiais que representam computações como adição e multiplicação. Os valores que o operador usa são chamados operandos.

Todas as expressões seguintes são válidas em Python e seus significados são mais ou menos claros:

```
20+32 hora-1 hora*60+minuto minuto/60 minuto//60
5**2 (5+9)*(15-7)
```

Em Python, os símbolos +, -, / e o uso de parênteses para agrupamento têm o mesmo significado que em matemática.

O asterisco (*) é o símbolo para multiplicação, ** é o símbolo para potenciação e // é o símbolo para divisão inteira.

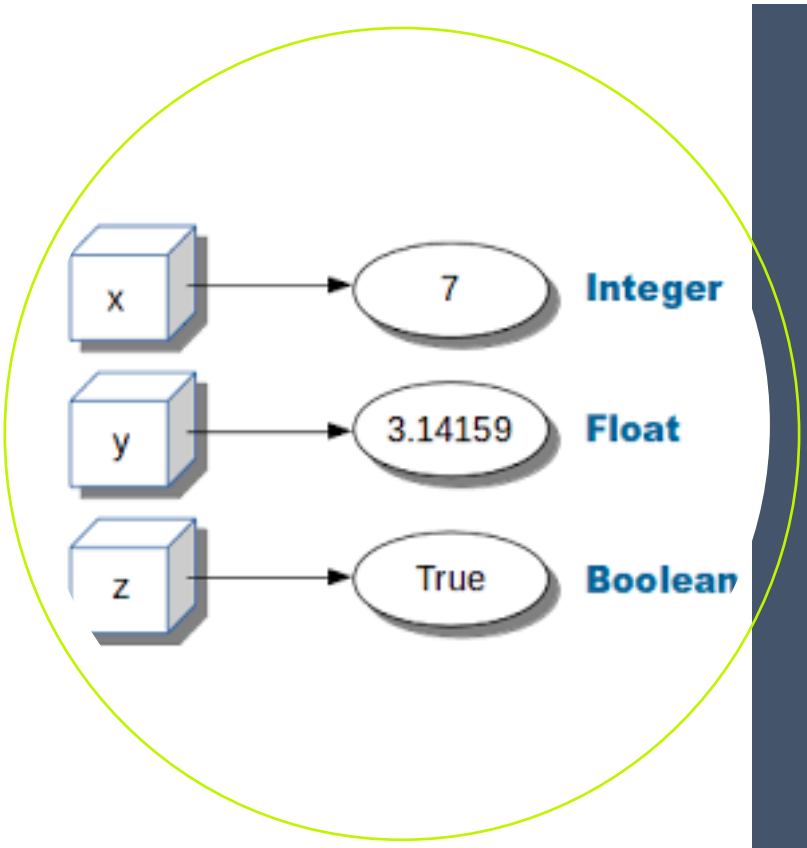
Quando um nome de variável aparece no lugar de um operando, ele é substituído pelo valor da variável, antes da operação ser executada.

Adição, subtração, multiplicação e potenciação fazem o que se espera: quando todos os operandos são inteiros, o resultado da operação é um valor inteiro.

Você pode ficar surpreso com a divisão. Observe as seguintes operações:

```
>>> minuto = 59

>>> minuto/60

0.98333333333333328

>>> minuto = 59

>>> minuto//60

0
```

O valor de minuto é 59 e, em aritmética convencional (/), 59 dividido por 60 é 0,98333. Já a divisão inteira (//) de 59 por 60 é 0.



COMPOSIÇÃO

Por outro lado, uma diferença significativa separa concatenação e repetição de adição e multiplicação. Você saberia mencionar uma propriedade da adição e da multiplicação que não ocorre na concatenação e na repetição?

Até agora, vimos os elementos de um programa (variáveis, expressões, e instruções ou comandos) isoladamente, sem mencionar como combiná-los. Uma das características mais práticas das linguagens de programação é a possibilidade de pegar pequenos blocos e combiná-los numa composição.

STRINGS

Para strings, o operador + representa concatenação, que significa juntar os dois operandos ligando-os pelos extremos.

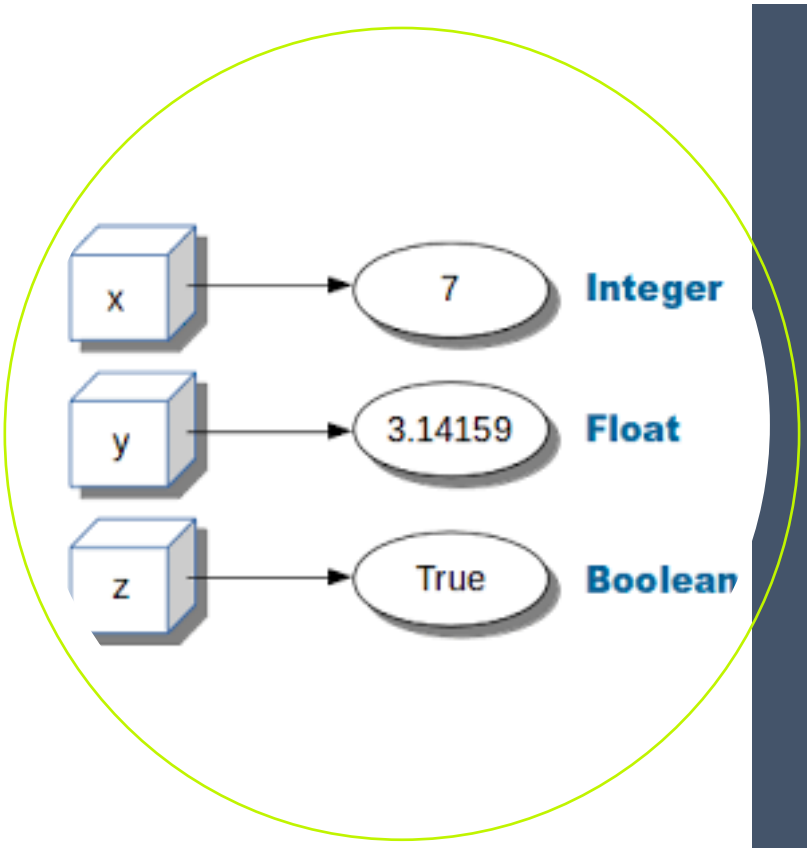
Por exemplo:

```
fruta = "banana"

assada = " com canela"

print (fruta + assada)
```

A saída deste programa é banana com canela. O espaço antes da palavra com é parte da string e é necessário para produzir o espaço entre as strings concatenadas.

O operador * também funciona com strings; ele realiza repetição. Por exemplo, "Legal"*3 é "LegalLegaLegal". Um dos operadores tem que ser uma string; o outro tem que ser um inteiro.

Por um lado, esta interpretação de + e * faz sentido pela analogia entre adição e multiplicação. Assim como 4*3 equivale a 4+4+4, não é de estranhar que "Legal"*3 seja o mesmo que "Legal"+"Legal"+"Legal".



Por exemplo, nós sabemos como somar números e sabemos como exibi-los; acontece que podemos fazer as duas coisas ao mesmo tempo:

```
>>> print (17 + 3)

20
```

Na realidade, a soma tem que acontecer antes da impressão, assim, as ações não estão na realidade acontecendo ao mesmo tempo. O ponto é que qualquer expressão envolvendo números, strings, e variáveis pode ser usada dentro de uma chamada da função print. Analise o exemplo ao lado.

ORDEM DE EXECUÇÃO

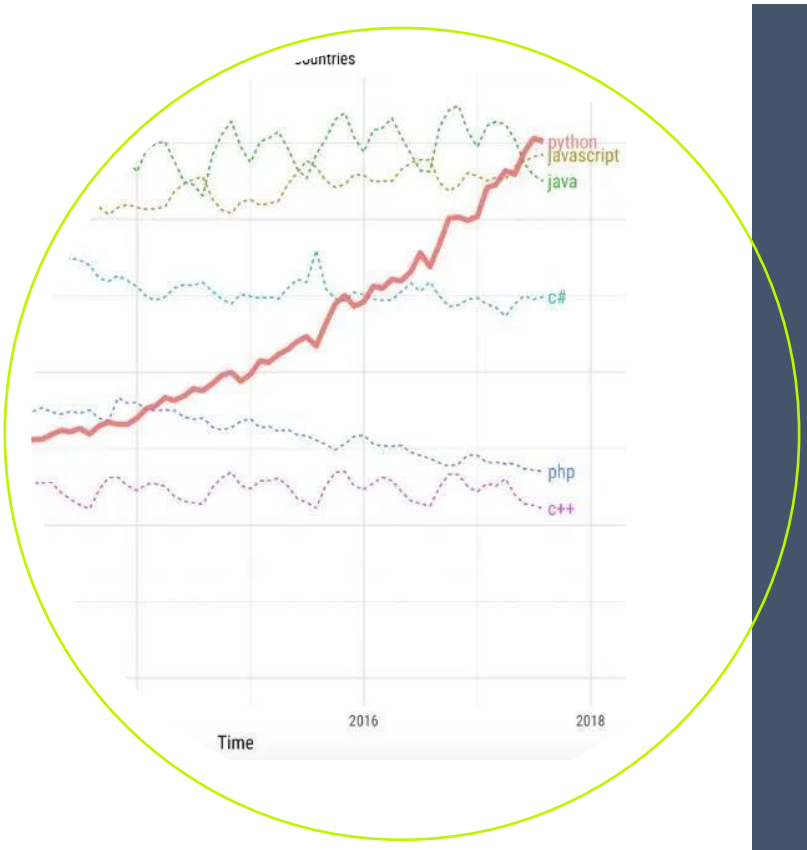
```
print ("Número de minutos desde a meia-noite: ", hora*60+minuto)
```

Esta possibilidade pode não parecer muito impressionante agora, mas você verá outros exemplos em que a composição torna possível expressar cálculos e tarefas complexas de modo limpo e conciso.

Atenção: Existem limites quanto ao lugar onde você pode usar certos tipos de expressão.

Por exemplo, o lado esquerdo de um comando de atribuição tem que ser um nome de variável, e não uma expressão. Assim, o seguinte não é válido:

```
minuto+1 = hora.
```



IDENTIFICADOR (MEMÓRIA)

Como outro exemplo, a função id recebe um valor ou uma variável e retorna um inteiro, que atua como um identificador único para aquele valor:

```
>>> id(3)

134882108

>>> bia = 3

>>> bia(beth)

134882108
```

Todo valor tem um id, que é um número único relacionado ao local onde ele está guardado na memória do computador. O id de uma variável é o id do valor a qual ela se refere.

FUNÇÕES

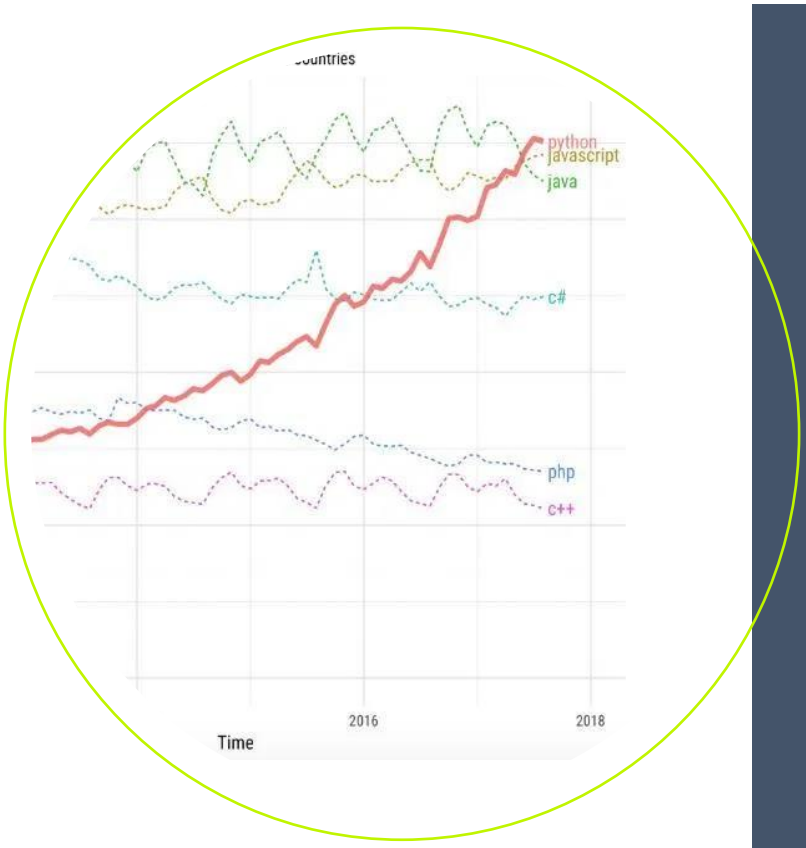
Você já viu um exemplo de uma chamada de função:

```
>>> type('32')

<class 'str'>
```

O nome da função é type e ela exibe o tipo de um valor ou variável. O valor ou variável, que é chamado de argumento da função, tem que vir entre parênteses. É comum se dizer que uma função 'recebe' um valor ou mais valores e 'retorna' um resultado.

O resultado é chamado de valor de retorno. Em vez de imprimir um valor de retorno, podemos atribui-lo a uma variável:

```
>>> bia = type('32')

>>> print (bia)

<class 'str'>
```



MAIS DE CONVERSÕES

A função float converte inteiros e strings em números em ponto flutuante:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finalmente, a função str converte para o tipo string:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Pode parecer curioso que Python faça distinção entre o valor inteiro 1 e o valor em ponto flutuante 1.0. Eles podem representar o mesmo número, mas pertencem a tipos diferentes. A razão é que eles são representados de modo diferente dentro do computador.

CONVERSÃO ENTRE TIPOS

Python provê uma coleção de funções nativas que convertem valores de um tipo em outro. A função int recebe um valor e o converte para inteiro, se possível, ou, senão, reclama:

```
>>> int('32')
32
>>> int('Alô')
ValueError: invalid literal for int() : Alô
```

Int também pode converter valores em ponto flutuante para inteiro, mas lembre-se que isso trunca a parte fracionária:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

ENTRADAS PELO TECLADO

Os programas que temos escrito até agora são um pouco crus, no sentido de não aceitarem dados entrados pelo usuário. Eles simplesmente fazem a mesma coisa todas as vezes.

Python fornece funções nativas que pegam entradas pelo teclado. A mais simples é chamada input. Quando esta função é chamada, o programa para e espera que o usuário digite alguma coisa.
Quando o usuário aperta a tecla Enter ou Return, o programa prossegue e a função input retorna o que o usuário digitou como uma cadeia de caracteres (string):

```
>>> entrada = input()
O que você está esperando?
>>> print (entrada)
O que você está esperando?
```

Antes de chamar input, é uma boa ideia exibir uma mensagem dizendo ao usuário o que ele deve entrar. Esta mensagem é como se fosse uma pergunta ou um alerta(prompt). Esta pergunta pode ser enviada como um argumento para input:

```
>>> nome = input("Qual... é o seu nome? ")
Qual... é o seu nome? Arthur, Rei dos Bretões!
>>> print (nome)
Arthur, Rei dos Bretões!
```

Se esperamos que a entrada seja um inteiro, podemos usar a função int aplicada à função input:

```
pergunta = "Qual... é a velocidade de vôo de uma andorinha?\n"
velocidade = int(input(pergunta))
```

Se o usuário digita uma cadeia de dígitos, ela é convertida para um inteiro e atribuída a velocidade. Infelizmente, se o usuário digitar um caractere que não seja um dígito, o programa trava:

```
>>> velocidade = int(input(pergunta))
Qual... é a velocidade de vôo de uma andorinha?
De qual você fala, uma andorinha Africana ou uma Europeia?
SyntaxError: invalid syntax
```

Para evitar esse tipo de erro, geralmente é bom usar input para pegar uma string e, então, usar funções de conversão para converter para outros tipos.