

Objetos e Comandos de Entrada e Saída em Python

Conceito de variáveis

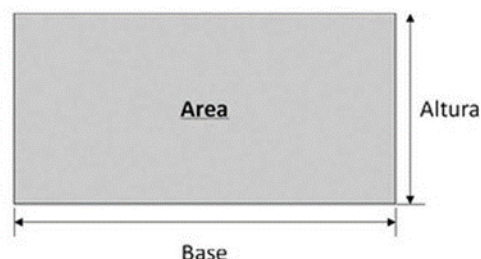
Todo algoritmo que se possa construir utilizará conjuntos de dados. Tais dados podem ser, basicamente, números e caracteres isolados ou, de algum modo, agrupados.

Para que um algoritmo possa ser implementado em um computador, é preciso que exista um meio de armazenamento dos dados que serão manipulados. Assim, chega-se ao conceito existente em todas as linguagens de programação e que é usualmente designado pelo termo “variável”.

Em programação de computadores, uma variável é um elemento da linguagem que ocupa um ou mais bytes na memória RAM do computador. Esse local da memória é capaz de reter, ou seja, armazenar o elemento de dado. No programa, a variável é identificada por um nome ou identificador. Assim, pode-se entender que do ponto de vista do programador a variável é um nome que contém um dado, e do ponto de vista do computador a variável é um endereço de memória que retém um conjunto de bits que representam esse dado.

Por exemplo, imagine que se queira escrever um algoritmo capaz de calcular a área de um retângulo. Nesse algoritmo, haverá três dados, sendo dois de entrada – a base e a altura do retângulo –, e terceiro, o resultado, é a área calculada utilizando-se os outros dois. Assim sendo, pode-se esquematizar um rascunho de algoritmo que seja o seguinte:

```
1. Obter o valor da Base
2. Obter o valor da Altura
3. Calcula Area = Base x Altura
4. Exibir a Area calculada
```



Um primeiro algoritmo utilizado para ilustrar o conceito de variável.

Os quatro passos sequenciais aqui exibidos representam um algoritmo simples, e os identificadores Base, Altura e Area são variáveis.

Modelo de dados de Python



No contexto de linguagens de programação, a expressão “modelo de dados” diz respeito à abordagem, aos paradigmas e às técnicas adotadas no projeto conceitual da linguagem, visando definir a maneira como os dados serão mantidos em memória e acessados pelo conjunto de instruções.

O modelo de dados do Python (*Python Data Model*, em inglês) adota como paradigma que todo dado em um programa escrito com Python é representado por um **objeto**. Todo objeto Python tem três aspectos: um identificador, um tipo e um conteúdo.

O **identificador** é o nome que o objeto tem no programa.

O **tipo** do objeto determina não só a natureza dos dados que este armazena (por exemplo, um número inteiro, um texto), mas também as operações que são suportadas por ele. Cada objeto em Python é criado a partir de uma classe (*class*), que é um elemento do paradigma de programação conhecida como programação orientada a objetos. Será visto, nos capítulos posteriores, que os objetos, além do conteúdo, apresentam comportamentos associados. Assim, um objeto do tipo “*int*” (número inteiro) terá associado a si um conjunto de funções adequadas à manipulação de números inteiros; um outro objeto do tipo “*list*” (lista) terá outras funções que se adequam à manipulação de listas; e assim por diante. O **conteúdo** do objeto é o valor (ou conjunto de valores) armazenado nele.

Exemplo 2.1 Objetos em Python

```
>>> MeuObjeto = 10
```

```
>>> type(MeuObjeto)
```

```
<class 'int'>
```

No Exemplo 2.1, podem ser vistos os três aspectos mencionados: o identificador é `MeuObjeto`; o tipo é “`int`”, que representa números inteiros; e o conteúdo é o valor 10.

Considerando o que foi exposto, a conclusão imediata é que em Python não existem **variáveis**, como estas costumam ser conhecidas em outras linguagens. O que existe, de fato, são os **objetos**. Pode parecer uma simples questão de nomenclatura, mas não se trata disso, uma vez que cada objeto, além de ser utilizado para armazenar seu conteúdo, apresenta um comportamento próprio associado à classe a que pertence.



Assim sendo, deste ponto em diante, será dada preferência ao uso do termo “objeto” em detrimento de “variável” para referência aos elementos relacionados ao armazenamento de dados em programas escritos com Python. A seguir, serão apresentados os tipos de objetos utilizados com maior frequência nos programas desenvolvidos por quem está iniciando o aprendizado de programação. Por questão de didática, será feita uma distinção entre tipos cujo conteúdo é indivisível, designados como “tipos simples”, daqueles cujos conteúdos representam coleções de elementos que podem ser acessados individualmente ou em grupo e que serão designados como “tipos estruturados”.

Tipos simples de dados

Embora nesse quesito haja muita semelhança entre as diversas linguagens existentes, cada uma tem suas peculiaridades. Em Python, estão disponíveis os tipos simples relacionados a seguir.

- **Número inteiro (*int*):** capazes de armazenar números inteiros positivos, zero ou negativos.
- **Número real (*float*):** capazes de armazenar números reais positivos ou negativos, além do zero. O separador decimal é o ponto “.”.
- **Número complexo (*complex*):** armazena um número complexo do tipo $4 + 3j$ (note-se, na parcela imaginária, que é utilizada a letra “j” em vez da letra “i”). A linguagem Python tem suporte completo às operações aritméticas envolvendo números complexos. Essa característica é muito útil em programas voltados à solução de problemas de física e engenharia, nos quais há uma forte presença de números complexos, por exemplo, estudo de vibrações, circuitos elétricos e sistemas dinâmicos.

Tipos estruturados de dados

Em contraposição aos tipos simples, os tipos estruturados são compostos, ou seja, seu conteúdo é constituído por outros elementos. Assim sendo, tais tipos representam agregados de objetos que podem ser acessados e manipulados em conjunto ou isoladamente.

Os tipos compostos mais importantes para esta fase do aprendizado de lógica de programação serão objeto de estudo específico do Capítulo 4 deste livro. Desse modo, a



proposta aqui é apenas listar e conceituar brevemente os tipos compostos existentes, dando uma visão geral das possibilidades da linguagem. O aprofundamento será estudado mais adiante.

- **Cadeia de texto (str):** também denominados strings, objetos deste tipo contêm qualquer sequência de caracteres, incluindo letras, algarismos e caracteres especiais. Servem para armazenar nomes, endereços, texto em geral, bem como quaisquer dados aos quais não se aplicam ou não se realizam operações aritméticas. Utilizando strings em Python, é possível acessar todo o texto ou cada caractere individualmente. Este é um tipo imutável, de modo que não é possível alterar um caractere isoladamente. O programador poderá ter acesso individual a ele, para exibi-lo na tela, por exemplo, porém não será capaz de alterá-lo. Se tentar fazê-lo, receberá uma mensagem de erro.

- **Lista (list):** de todos os tipos disponíveis em Python, é um dos mais versáteis e poderosos. Uma lista caracteriza-se por ser um conjunto de itens entre colchetes e separados por vírgulas. Os itens não precisam ser todos do mesmo tipo e podem ser acessados e manipulados individualmente, todos de uma vez ou em grupos. Este é um tipo mutável.

- **Tupla (tuple):** são semelhantes às listas, no entanto, seus componentes não podem ser alterados. Este é um tipo imutável. Uma tupla caracteriza-se por ser um conjunto de itens entre parênteses e separados por vírgulas.

- **Conjunto (set e frozenset):** um conjunto é uma coleção não ordenada de elementos não duplicados e caracteriza-se por itens entre chaves e separados por vírgulas. Tem diversos usos possíveis e suporta operações matemáticas típicas de conjuntos, como união, interseção e diferença, muito úteis em alguns algoritmos. O tipo set é mutável, enquanto que frozenset é imutável.

- **Dicionário (dict):** também designado como tipo mapeado, um dicionário é uma coleção de pares “chave:valor” não ordenados, com a obrigatoriedade de que as chaves sejam únicas (não duplicadas) dentro do dicionário. Enquanto as listas e tuplas são indexadas por um número inteiro, os dicionários são indexados pela chave associada ao valor. Dicionários são mutáveis.

Por fim, uma breve palavra sobre um conceito muito relevante em Python que está relacionado aos tipos de dados e com frequência deixa confuso o programador que já conhece



outras linguagens e está iniciando seus estudos de Python.

Os objetos existentes na linguagem podem ser classificados como **imutáveis** (*immutable*) ou **mutáveis** (*mutable*). Um objeto imutável tem conteúdo fixo, ou seja, não pode ser alterado sem que o objeto seja reconstruído. Os tipos numéricos (int, float, complex), os strings, tuplas e frozenset são imutáveis, de modo que, quando um novo conteúdo for atribuído ao objeto, sua instância anterior é removida, e uma nova instância, criada. Os tipos lista, dicionário e set são mutáveis, de modo que podem ter seu conteúdo alterado, sem que sua instância seja recriada.

Esse conceito é realmente relevante em Python e será mais bem detalhado no Item 2.3.2. Ao programador iniciante a sugestão é que não se preocupe com esse assunto agora e se aprofunde mais no aprendizado da linguagem. Para os programadores experientes a sugestão é que não desistam do Python, pois no devido momento esse conceito, e diversos outros, muito típicos do Python, serão compreendidos e farão todo o sentido. Para ambos, convém dizer que será muito gratificante conhecer os detalhes e paradigmas do Python, pois são muito bem pensados e implementados.

Começando a trabalhar

Agora, é hora de começar a trabalhar com Python. Para isso, supondo que já o tenha instalado, abra o ambiente IDLE e digite os comandos deste programa:

Exemplo 2.2 Um primeiro programa

```
>>> Base = 10
>>> Altura = 4
>>> Area = Base * Altura
>>> print(Area) 40
```

Os quatro comandos do Exemplo 2.2 são a tradução para Python do algoritmo da Figura 2.1. Os dois primeiros comandos atribuem valores fixos aos objetos Base e Altura. O terceiro comando contém, do lado direito, uma expressão aritmética de multiplicação cujo resultado é calculado e armazenado no objeto Area. Por fim, o comando print é utilizado para exibir na tela o conteúdo de Area, no caso, 40.

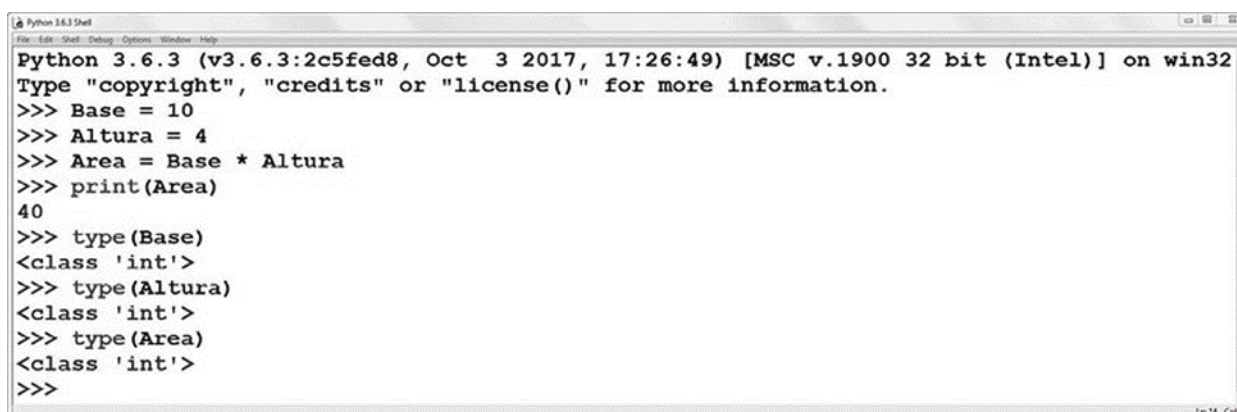
Nesse pequeno exemplo já fica evidente um aspecto importante do Python. Os objetos do



programa não precisam ser explicitamente declarados para serem utilizados. Essa declaração explícita é necessária na maioria das linguagens, mas em Python, não. É frequente que esse aspecto da linguagem cause estranheza em quem já conhece alguma outra linguagem, como C, C++ ou Java.

Para que um objeto comece a existir, basta que a ele se atribua um valor inicial. Ao fazer isso, o Python cria seu identificador e reserva um espaço de memória para armazenar o dado contido.

Outra questão que surge como decorrência desse processo de criação do objeto a partir da atribuição de valor inicial é quanto ao tipo, ou classe, do objeto. No caso desse exemplo, todos são números inteiros, formalmente: são do tipo "int". Para constatar isso, utilize o comando *type*. Veja a Figura 2.2, em que foi utilizado o comando *type* três vezes, com o qual se verifica que, de fato, os objetos criados são do tipo "int".



```
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Base = 10
>>> Altura = 4
>>> Area = Base * Altura
>>> print(Area)
40
>>> type(Base)
<class 'int'>
>>> type(Altura)
<class 'int'>
>>> type(Area)
<class 'int'>
>>>
```

Uso do comando *type*.

Caso se queira criar um objeto do tipo "float", ou seja, capaz de conter um número real, basta atribuir a ele um valor que contenha a parte decimal.




```
Python 3.6.3 Shell
Python 3.6.3 (v3.6.3:2c5fed8, Oct 3 2017, 17:26:49) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> Base = 10
>>> Altura = 4
>>> Area = Base * Altura
>>> print(Area)
40
>>> type(Base)
<class 'int'>
>>> type(Altura)
<class 'int'>
>>> type(Area)
<class 'int'>
>>> Base = 10.2
>>> Altura = 4.0
>>> Area = Base * Altura
>>> print(Area)
40.8
>>> type(Base)
<class 'float'>
>>> type(Altura)
<class 'float'>
>>> type(Area)
<class 'float'>
>>>
```

Uso do comando `type`.

No exemplo da Figura 2.3 foram utilizados, propositalmente, os mesmos nomes para os objetos e, em seguida, foi utilizado o comando `type` três vezes para verificar o tipo de cada um. Perceba que, se antes eles eram “int”, após o uso com números reais eles passaram a ser “float”. No momento de atribuição de um valor ao objeto, o interpretador verificará qual tipo mais adequado o utilizará.

Na linguagem Python, essa é uma característica relevante. Ao iniciante parece que os objetos podem mudar de tipo sempre que houver uma atribuição de valor. Na prática, o que ocorre é algo diferente: a cada operação de atribuição um novo objeto é criado em memória, sendo o anterior descartado.

Cada objeto criado tem uma **identidade** (*identity*), que é um número inteiro criado no momento em que ocorre a atribuição de valor.

Isso pode ser constatado por meio do uso do comando `id`, que retorna a identidade do objeto. Observe a Figura 2.4, na qual o mesmo nome foi utilizado em três atribuições consecutivas. Cada uma das atribuições criou um objeto com identidade diferente, sendo que o nome identificador permaneceu o mesmo. Essa característica confere grandes flexibilidade e poder à programação Python. No momento, faltam elementos para comprovar tal afirmação, mas isso será mostrado mais adiante.

É importante não confundir dois termos utilizados até aqui: identificador e identidade. O



primeiro é o nome do objeto utilizado ao se escrever o programa, e o segundo é um número inteiro criado quando o programa está em execução.



```
>>> MeuObj = 25
>>> id(MeuObj)
1456855216
>>> MeuObj = 3.7
>>> id(MeuObj)
47259360
>>> MeuObj = 'Teste'
>>> id(MeuObj)
50236864
>>>
```

Verificação da identidade de um objeto.

Agora, faça uma experimentação. Abra o IDLE em seu computador e experimente as sugestões a seguir. Utilize a função **id()** também.

Exemplo 2.3 Teste o que já aprendeu

```
>>> X = 1.0
>>> type(X) # X é float
>>> Y = 18
>>> type(Y) # Y é int
```

```
>>> Z = X + Y
```

```
>>> type(Z) # Z é a soma de float com int. Qual é o tipo de Z?
```

```
>>> a = 5 + 3j
```

```
>>> type(a) # a é complex
```




```
>>> type(A) # Experimente usar o type com o objeto A (maiúsculo) # e verifique o que ocorre.
```

Nomes de objetos: identificadores

Quando o programador estiver escrevendo um programa, ele precisará atribuir nomes aos objetos, ou seja, definir o identificador que utilizará em cada um. Todas as linguagens têm regras para o estabelecimento desses identificadores.

É claro que tais regras vão variar de uma linguagem para outra, mas, em linhas gerais, os identificadores podem ser criados usando letras, números e o caractere *underscore* “_” (também denominado *underline*).

Em algumas linguagens, como Object Pascal, não faz diferença utilizar letras minúsculas ou maiúsculas, ou seja, os identificadores “Valor” e “valor” serão tratados como a mesma coisa e farão referência ao mesmo endereço de memória.

Em outras linguagens, como C, Java e Python, isso faz total diferença. Nesses casos, os identificadores “Valor” e “valor” vão se referir a diferentes endereços na memória do computador.

Em Python, os identificadores devem começar com uma letra, que pode ser aiúscula ou minúscula, ou com o caractere *underscore*. Não é permitido que um identificador comece com um número.

É recomendável que os objetos sejam criados com nomes que ajudem a lembrar seu conteúdo. No Exemplo 2.2, os identificadores Altura, Largura e Area poderiam ser substituídos por X, Y e Z, por v1, v2, v3 ou qualquer outra coisa, e ainda assim o programa funcionaria do mesmo modo. Porém, utilizar nomes que facilitam lembrar o que o objeto contém gera um ganho de produtividade durante o desenvolvimento do programa e, principalmente, facilita muito nas manutenções e atualizações futuras, quando é preciso lembrar o que o programa faz e como faz, para que as modificações possam ser desenvolvidas.

Atribuições e expressões aritméticas

Atribuições

Uma operação de atribuição, ou simplesmente “Atribuição”, já foi utilizada nos exemplos anteriores, e trata-se de uma expressão envolvendo o operador de atribuição “=”. Observe a linha a seguir:

Destino = Origem



Essa linha refere-se a uma atribuição, na qual Destino é um identificador de objeto e Origem podem ser diversos elementos, tais como um valor, um objeto, uma expressão aritmética, o retorno de uma função etc. Trata-se de algo simples de ser compreendido, porém, nos bastidores dessa operação simples estão implementados alguns conceitos e características importantes que serão agora descritos.

Em primeiro lugar, as atribuições criam um objeto e o associam a um nome identificador, conforme mostrado nas Figuras 2.2 a 2.4. É importante olhar primeiro para o lado direito da atribuição, pois é a expressão ali contida que define o tipo de objeto que será criado. Uma vez avaliada a expressão e criado o objeto em memória, o identificador definido do lado esquerdo é associado ao objeto, como se uma etiqueta fosse colocada em uma peça produzida. Esse é o principal conceito de bastidor envolvendo as atribuições.

Os nomes de identificadores são criados quando atribuídos pela primeira vez. Caso sejam atribuídos novamente, há duas situações a considerar. Se a natureza do objeto é imutável, então, a instância anterior é destruída e uma nova instância é criada. É por esse motivo que, na Figura 2.4, a cada atribuição um novo *id* é criado. Porém, se a natureza do objeto for mutável, então, o objeto será mantido em memória e a alteração de seu conteúdo será feita sem mudança de *id*. Isso tem implicações que, se não forem bem compreendidas pelo programador Python, podem causar um entendimento errôneo do que está acontecendo em um programa. No Capítulo 4, serão descritas algumas situações em que isso é relevante.

Outro ponto importante: antes de serem referenciados em qualquer comando ou expressão, os identificadores precisam ser criados. Utilizar um identificador sem antes criá-lo com uma atribuição fará que o interpretador gere um erro e interrompa o programa.

Formas de atribuição

O Exemplo 2.4 exhibe um grupo contendo as mais simples e frequentes formas de atribuição utilizadas. Embora simples, o que é feito nos bastidores não é óbvio. Observe a criação do objeto B no exemplo. A atribuição $B = A$ faz que o identificador B passe a apontar para o mesmo objeto em memória para o qual aponta o objeto A. O conceito de bastidor aqui envolvido parte da ideia de que, se dois identificadores devem ter o mesmo conteúdo, então, é razoável que ambos apontem para o mesmo objeto, promovendo um melhor uso da memória. Logo em seguida, é atribuído o valor 50 a B, e neste caso um novo objeto é criado em memória e o



identificador B passa a apontar para ele, tendo seu *id* alterado. Esse comportamento ocorre porque os objetos A e B, do tipo “int”, são imutáveis.

O termo **imutável** já foi mencionado no item 2.1.4, e agora estão disponíveis os elementos para explicá-lo convenientemente. Quando um objeto é imutável e seu conteúdo é trocado, o objeto anterior é descartado e um novo é criado. No Exemplo 2.4, isso é mostrado por meio da verificação do *id* do objeto B antes e depois de receber o novo valor. Em contrapartida, um objeto **mutável** poderá ter seu conteúdo livremente alterado, ao mesmo tempo que seu *id* é mantido. Talvez o programador experiente agora compreenda melhor o que está acontecendo, mas talvez ainda não entenda por que variáveis simples, como um “int”, tenham de ser descartadas e recriadas a cada nova atribuição. Bem, então, o próximo passo no aprofundamento desse assunto será dado no início do Capítulo 6.

Seguindo no assunto de atribuições, observe-se o que ocorre com os identificadores L e M, que apontam para objetos do tipo lista. As listas são mutáveis de modo que uma alteração nos elementos contidos na lista não provoca a criação de um novo *id* e, por consequência, os dois identificadores L e M continuam apontando para o mesmo objeto que foi alterado. As listas são tipos sequenciais a serem estudados no Capítulo 4.

É frequente que programadores experientes em outras linguagens fiquem confusos com esse comportamento de Python, que lhes parece produzir resultados inesperados. Ao contrário de inesperado, estes são sólidos conceitos implementados em Python, e cabe ao programador interessado nessa linguagem buscar conhecê-los para poder fazer bom uso deles.

Exemplo 2.4 Formas simples de atribuição

```
>>> A = 10 # A é criado e recebe o valor 10
```

```
>>> id(A)
```

```
498390976 # A tem um id
```

```
>>> B = A # B é criado, recebendo A
```

```
>>> id(B) # observe que o id de B é o mesmo que o id de A 498390976 # foi criado o novo nome (B) que aponta para A
```



```
>>> B = 50 # nova atribuição para B
```

```
>>> id(B) # que passa a ter um novo id 498391616
```

```
>>> L = [12, 24, 36] # cria a lista L
```

```
>>> id(L) 48917320
```

```
>>> M = L # cria a lista M que passa a ter o mesmo id de L
```

```
>>> id(M) 48917320
```

```
>>> M[0] = 0 # altera-se um elemento de M
```

```
>>> print(M)
```

```
[0, 24, 36]
```

>>> print(L) # o elemento de L também foi alterado. Isto [0, 24, 36] # ocorre porque listas são mutáveis.

```
>>> C = A * 2
```

```
>>> id(C) 498391136
```

```
>>> D = "TEXTO" # o objeto apontado por D é um tipo estruturado
```

```
>>> type(D) # string. A manipulação de strings em Python
```

```
<class 'str'> # é muito simples e poderosa
```

```
>>> from math import sqrt
```

```
>>> X = 25
```

```
>>> Y = sqrt(X) # o retorno de uma função também cria objetos
```



```
>>> Y 5.0
```

Continuando com o Exemplo 2.4, na criação do objeto C foi utilizada uma expressão aritmética envolvendo o objeto A e um valor. Em seguida, foi criado o objeto D com a atribuição de um string. Assim como as listas, os strings são tipos sequenciais, os quais serão estudados no Capítulo 4.

Por fim, foi feita a importação da função *sqrt* – raiz quadrada – do módulo *math*, a qual foi utilizada para criar o objeto Y. Veja o Item 2.4 para mais informações sobre funções matemáticas.

O Python ainda suporta outros tipos de atribuição, mostradas no Exemplo 2.5. O caso 1 é o de atribuição múltipla, em que vários identificadores são criados simultaneamente. Se você já compreendeu os conceitos de bastidores implementados em Python deverá raciocinar que A, B e C são três identificadores distintos que apontam para o mesmo objeto em memória e, portanto, têm o mesmo *id*. E esse raciocínio está correto, como pode ser constatado no exemplo.

O caso 2 exemplifica a atribuição posicional. Cada objeto criado no lado direito da expressão é atribuído a cada identificador do lado esquerdo, segundo a posição relativa de cada um, de modo que A = 1, B = 2 e C = 3. Nesse caso, são objetos diferentes, portanto, cada identificador tem o próprio *id*. No Capítulo 4, esse tipo de atribuição será retomado, uma vez que essa operação envolve uma tupla de identificadores do lado esquerdo e uma tupla de objetos do lado direito da expressão.

O caso 3 também é uma atribuição de tuplas. Com essa forma de atribuição, é possível inverter os conteúdos de dois objetos. Essa é uma situação comum em muitos algoritmos, e em outras linguagens exige que uma variável intermediária seja utilizada na troca. Em Python, basta escrever essa forma de atribuição e os conteúdos serão invertidos. Esse caso pode ser generalizado para qualquer quantidade de objetos envolvidos.

Exemplo 2.5 Outros tipos de atribuição em Python

```
>>> A = B = C = 1 # caso 1: atribuição múltipla
```

```
>>> id(A) # A, B e C tem o mesmo id 498390832
```

```
>>> id(B) 498390832
```

```
>>> id(C) 498390832
```



```
>>> A, B, C = 1, 2, 3 # caso 2: atribuição posicional
```

```
>>> id(A) 498391008
```

```
>>> id(B) 498390848
```

```
>>> id(C) 498390864
```

```
>>> X, Y, Z = 0, -10, 10
```

```
>>> print(X, Y) 0 -10
```

```
>>> X, Y = Y, X # caso 3: inversão de objetos
```

```
>>> print(X, Y)
```

```
-10 0
```

```
>>> print(X, Y, Z)
```

```
-10 0 10
```

```
>>> X, Y, Z = Y, Z, X # é possível generalizar este caso
```

```
>>> print(X, Y, Z) # para qualquer quantidade de 0 10 -10 # objetos envolvidos
```

Expressões aritméticas

As expressões aritméticas são construídas utilizando-se objetos, operadores aritméticos e funções matemáticas, sendo que toda linguagem de programação permite a construção de operações aritméticas. Uma expressão aritmética é algo do tipo

$$R = A + B$$

em que: A e B são objetos numéricos e R recebe o resultado de sua adição. Nessa expressão, A e B são chamados de operandos e “+” é o operador aritmético de adição.

Como vimos, em Python estão disponíveis três tipos numéricos: inteiros, reais e



complexos. Com esses três tipos é possível realizar operações aritméticas.

É possível misturar objetos de diferentes tipos numéricos em uma única expressão. Quando houver uma situação assim, o interpretador Python buscará a melhor maneira de resolvê-la. Havendo, em uma expressão, a mistura de operandos inteiros e reais, o resultado calculado será real. E quando houver inteiros, reais e complexos, o valor resultante será tratado como complexo.

Os operadores aritméticos disponíveis em Python são os indicados no Quadro 2.1. Execute todos os exemplos da tabela com os valores: $A = 14$ e $B = 5$. Obedeça ao esquema a seguir e compare os resultados que você obteve com os resultados esperados indicados no Quadro 2.1.

Operação	Operador	Exemplo	Resultado esperado
Adição	+	$C = A + B$	19
Subtração	-	$C = A - B$	9
Multiplicação	*	$C = A * B$	70
Divisão	/	$C = A / B$	2.8
Divisão inteira	//	$C = A // B$	2
Resto (módulo)	%	$C = A \% B$	4
- unário	-	$C = -A$	-14
Potenciação	**	$C = A ** B$	537824

Quadro 2.1 - Operadores aritméticos em Python.



```
Python 3.6.3 Shell
File Edit Shell Debug Options Window Help
>>> A = 14
>>> B = 5
>>> C = A + B
>>> print(C)
19
>>> C = A - B
>>> print(C)
9
>>> C = A * B
>>> print(C)
70
>>> C = A / B
>>> print(C)
2.8
>>> C = A // B
>>> print(C)
2
>>> C = A % B
>>> print(C)
4
>>> C = A ** B
>>> print(C)
537824
>>>
```

Uso dos operadores aritméticos.

É importante ressaltar que na linguagem Python, praticamente todos os operadores apresentados no Quadro 2.1 estão disponíveis para serem utilizados com os três tipos numéricos definidos na linguagem: int, float e complex.

Há duas exceções, no entanto, que são os operadores de cálculo de divisão de inteiros e resto que não estão definidos para os tipos complexos.

Construção de expressões aritméticas com múltiplos operadores

Em programação, é comum precisar escrever expressões aritméticas envolvendo dois ou mais operadores aritméticos. Nesses casos, a ordem de prioridade entre os operadores deve ser observada. O operador de maior prioridade sempre será calculado antes. Na expressão a seguir, primeiro será calculada a multiplicação entre 2 e A, e ao resultado será adicionado B.

$$R = 2 * A + B$$

Onde for necessário, pode-se alterar a prioridade das operações utilizando parênteses de maneira apropriada. Assim, se o desejado para essa expressão fosse somar A e B primeiro e multiplicar o resultado dessa soma por 2 em seguida, então, a expressão deve ser escrita como:

$$R = 2 * (A + B)$$

Em programação, em qualquer linguagem, incluindo Python, as regras de precedência da



álgebra são estritamente respeitadas e, se for preciso, pode-se abrir tantos parênteses quanto necessário.

Uma expressão aritmética muito utilizada nos algoritmos é aquela em que se toma o conteúdo de um objeto e a ele se soma (ou dele se subtrai etc.) um valor ou outra variável. Para efetuar uma operação, assim se escreve:

```
A = A + 1
```

Em Python, nestes casos, pode-se utilizar a operação de atribuição incremental, e a expressão ficará assim:

```
A += 1
```

Mais opções dessa operação estão exemplificadas no Exemplo 2.6.

Exemplo 2.6 Usos da atribuição incremental

```
>>> A = 10
```

```
>>> A 0
```

```
>>> A += 1 # atribuição incremental: adição 11
```

```
>>> A -= 5 # atribuição incremental: subtração
```

```
>>> A 6
```

```
>>> A *= 2 # atribuição incremental: multiplicação
```

```
>>> A 12
```

```
>>> A /= 4 # atribuição incremental: divisão
```

```
>>> A 3.0
```

```
>>> A = 10
```

```
>>> P = 4
```

```
>>> A += P # todas essas operações também
```



```
>>> A # podem ser feitas usando um objeto  
14 # no lugar do valor literal
```

Funções matemáticas

Junto com os operadores mostrados no Exemplo 2.6, em Python pode-se utilizar uma gama muito grande de funções matemáticas. Parte dessas funções está na biblioteca-padrão (em inglês, denominada pelo termo *built-in*), e outra parte está nas bibliotecas de funções “math” e “cmath”, que fornecem ao programador uma grande variedade de funções matemáticas prontas.

A biblioteca-padrão está sempre disponível, e não é necessário utilizar nenhum comando específico para carregá-la.

A biblioteca “math” contém funções que suportam apenas tipos inteiros e reais e precisa ser carregada para ser utilizada.

A biblioteca “cmath” contém funções que suportam tipos complexos e precisa ser carregada para ser utilizada.

Para conhecer todas as funções da biblioteca-padrão consulte a seção 2 da referência “*The Python Standard Library*”, cujo caminho em *Python Docs* é:

Python » Documentation » The Python Standard Library » 2. Built-in Functions

Para conhecer todas as funções suportadas em math e cmath, consulte as Seções 9.2 e 9.3, respectivamente, da referência “*The Python Standard Library*” cujo caminho em *Python Docs* é:

Python » Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules

Para utilizar tais bibliotecas, é necessário, primeiro, carregar a biblioteca desejada por meio do comando “from ... import ...”

```
>>> from math import sqrt  
>>> x = 9
```

```
>>> r = sqrt(x)
```

```
>>> print(r) 3.0
```



O Quadro 2.2 relaciona algumas funções matemáticas importantes, indicando o que fazem e a qual biblioteca pertencem. Essa lista é um subconjunto do que existe. Consulte as referências indicadas para conhecer tudo o que está disponível.

Função	Descrição	Observação
abs(x)	Valor absoluto (módulo) de x.	Bib. padrão
int(x)	Converter x para inteiro eliminando sua parte decimal. O conteúdo de x deve ser real.	Bib. padrão
float(x)	Converte x para número real. O conteúdo de x deve ser inteiro.	Bib. padrão
round(x [, n])	Arredonda x com n dígitos decimais. Se n for omitido, o valor 0 é assumido.	Bib. padrão
trunc(x)	O valor x é truncado, ou seja, a parte decimal é eliminada. Na prática, equivale ao int(x).	Bib. math
floor(x)	Retorna o maior inteiro $\leq x$.	Bib. math
ceil(x)	Retorna o menor inteiro $\geq x$.	Bib. math
sqrt(x)	Calcula a raiz quadrada de x.	Bib. math e cmath

Função	Descrição	Observação
exp(x)	Retorna o exponencial de x, ou seja, e^x .	Bib. math e cmath
log (x[, base])	Retorna o logaritmo de x na base fornecida. Se a base for emitida, calcula o logaritmo natural.	Bib. math e cmath

Quadro - Lista de funções matemáticas.

Comando de exibição – print

O propósito do comando print é a exibição na tela de qualquer informação relevante ao usuário do programa. Pode-se raciocinar em termos de que o print é o mais básico comando existente para que o programa “se comunique” com quem o está utilizando.



Exemplo 2.7 Uso do comando print

```
>>> print("Este é o Capítulo 2 do livro") # caso 1 Este é o Capítulo 2 do livro
```

```
>>> A = 12
```

```
>>> print(A) # caso 2 12
```

```
>>> B = 19
```

```
>>> print(B) # outro caso 2 19
```

```
>>> print(A, B) # caso 3 12 19
```

```
>>> print("Valor de A =", A) # caso 4 Valor de A = 12
```

```
>>> print("Valor de A = {0} e valor de B = {1}".format(A, B)) # c.5 Valor de A = 12 e valor de B = 19
```

Com o print, é possível mostrar mensagens de texto, conteúdos de objetos ou uma combinação das duas coisas, como pode ser visto no Exemplo 2.7.

Nesse exemplo, o print do caso 1 exibe uma mensagem de texto. Note que o texto deve ser escrito entre aspas, que podem ser duplas ("") ou simples (' '), para ser exibido como mensagem. A escolha do tipo de aspas cabe ao programador, é uma questão de gosto pessoal. O Python interpretará os dois tipos de aspas de maneira totalmente equivalente. Porém, é importante que não os misture, ou seja, se iniciou o texto com um tipo, deve finalizá-lo com ele.

Os prints identificados como caso 2 exibem um objeto cada um. Nesse caso, a diferença é que o nome do objeto é colocado no print sem o uso de aspas, e o que é exibido na tela é o conteúdo do objeto.

No print do caso 3, são exibidos simultaneamente os conteúdos de dois objetos. Isso faz que os valores sejam exibidos na mesma linha separados por um espaço em branco. Em casos assim, é possível alterar o caractere separador especificando-se um ou mais caracteres alternativos por meio do parâmetro *sep*, como mostrado no Exemplo 2.8.

Exemplo 2.8 Uso do comando print com separador

```
>>> print(A, B, sep="-") 12-19
```

```
>>> print(A, B, sep=", ") 12, 19
```

No print do caso 4, é exibida uma mensagem seguida do conteúdo de um objeto, e como



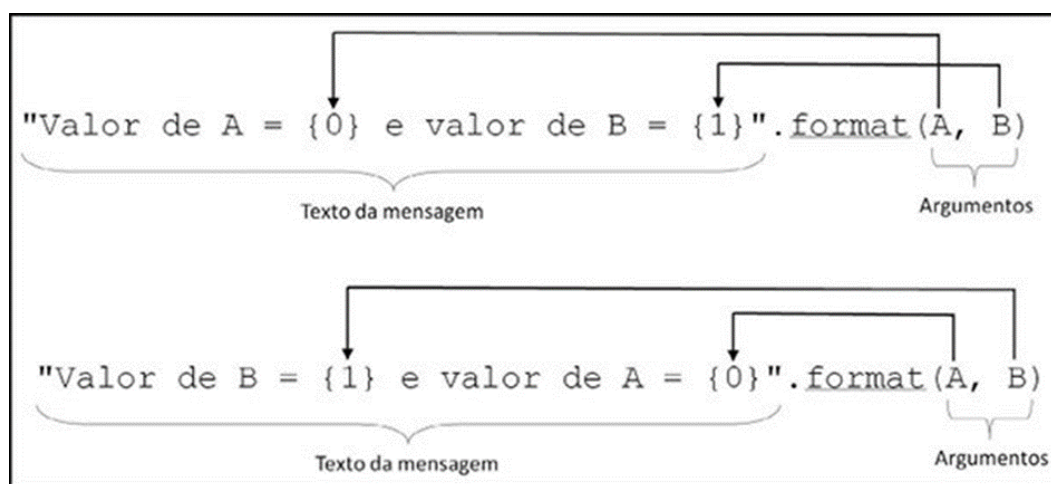
o parâmetro `sep` não foi especificado, foi inserido o espaço em branco padrão.

Por fim, no print do caso 5 do Exemplo 2.7, é mostrado como produzir uma saída formatada. Esse tipo de saída é muito útil para produzir exibições nas quais é possível controlar diversos detalhes dos elementos envolvidos.

Para produzir uma saída formatada, o primeiro passo é escrever a mensagem que se quer ver na tela, tomando o cuidado de utilizar os identificadores `{0}`, `{1}`, `{2}` etc. nos pontos da mensagem onde se deseja que apareça o conteúdo dos objetos envolvidos. O texto da mensagem deve ser seguido do método `format`, que conterà como argumentos os objetos que fornecerão os valores que substituirão os identificadores entre chaves.

A substituição dos identificadores pelos argumentos é feita seguindo-se o índice numérico, ou seja, nesse exemplo o conteúdo do objeto A substituirá o identificador `{0}`, porque A é o primeiro argumento, e o conteúdo de B substituirá o identificador `{1}`, independentemente do local em que esses identificadores estejam posicionados no texto.

É possível omitir o número dentro das chaves dos identificadores e utilizar apenas `{}`. Nesse caso, a associação entre identificador e objeto será feita pela ordem de ocorrência.



Exemplo de mensagem formatada.

Adicionalmente, os identificadores podem receber qualificadores de formatação que determinam como os dados devem ser apresentados. Isso se faz acrescentando `:"`, um caractere de formatação e o tamanho, ficando assim: `{0:d}` ou `{0:6.2f}` no caso de identificadores numerados; e `{:d}` ou `{:6.2f}` no caso de numeração omitida.

Também é possível especificar se o dado será alinhado à esquerda, à direita ou



centralizado, utilizando-se, respectivamente, os caracteres “<”, “>”, “^”.

Os tipos disponíveis são muito amplos, e no Quadro 2.3 são apresentados alguns de uso mais frequente. Para que se conheçam todas as opções com todos os detalhes existentes, é necessário recorrer à documentação oficial do Python referente à formatação de strings (disponível em <<https://docs.python.org/3.6/library/string.html>>).

Formatação	Resultado	Descrição



"Dado = {0:d}". format(A)	Dado = 9	d – número inteiro, em base 10.
"Dado = {0:5}". format(A)	Dado = 9	5d – número inteiro ocupando no mínimo 5 caracteres alinhado à direita.
"Dado = {0:f}". format(X)	Dado = 4.860000	f – número real, exibindo o padrão de 6 casas após a vírgula.
"Dado = {0:2f}". format(X)	Dado = 4.86	f – número real, exibindo 2 casas após a vírgula.
"Dado = {0:6.3f}". format(X)	Dado = 4.860	f – número real, ocupando no mínimo 6 caracteres e exibindo 1 casa após a vírgula.
"qq{:7d}qq". format(A)	qq 9qq	7d – número inteiro ocupando no mínimo 7 caracteres alinhado à direita.
"qq{:<7d}qq". format(A)	qq9 qq	7d – número inteiro ocupando no mínimo 7 caracteres alinhado à esquerda.
"qq{:^7d}qq". format(A)	qq 9 qq	7d – número inteiro ocupando no mínimo 7 caracteres centralizado.

Quadro - Formatação de exibição em tela.

Dica

Existe uma forma alternativa de trabalhar com strings formatados em Python. Ao fazer buscas pela internet, é muito provável que se depare com essa outra forma, que é muito parecida, porém, é diferente. Observe com atenção as duas linhas a seguir, sabendo de antemão que ambas produzem exatamente o mesmo resultado.

```
print("Valor de A = {0} e valor de B = {1}".format(A, B)) print("Valor de A = %d e valor de B = %d" % (A, B))
```

Nessa segunda opção utiliza-se "%d" no lugar dos identificadores {0} e {1}. E no lugar do método forma" utiliza-se o operador "%".

Essa forma assemelha-se muito ao modo como a linguagem C e algumas outras linguagens formatam suas saídas. Por que utilizar a primeira forma, então? A resposta encontra-se na documentação oficial do Python, na qual é declarado que se trata de uma forma obsoleta e que pode não ser suportada no futuro. Para saber,



**Python » Documentation » The Python Standard Library » 2.
Text Sequence Type » printf-style String Formatting**

Comando de entrada de dados – input

Toda linguagem de programação apresenta um ou mais comandos relacionados à entrada de dados por meio do teclado. Tal tipo de comando tem como propósito permitir que o usuário digite o dado de entrada no teclado.

Em Python 3, o comando para isso é o *input*. Esse comando tem um parâmetro string opcional que é exibido na tela antes de iniciar a leitura, a qual, uma vez iniciada, será concluída ao pressionar a tecla *Enter*. O que tiver sido digitado é carregado em um objeto de destino. Caso o objeto de destino não exista, será criado nesse momento. Sua forma de uso é mostrada no Exemplo 2.9, em que o objeto “x” é criado e recebe o retorno do input.

Exemplo 2.9 Uso do comando input

```
>>> x = input("Digite algo: ") Digite algo: teste de digitação
>>> x
```

```
'teste de digitação'
```

```
>>> n = input("Digite um número inteiro: ")
Digite um número inteiro: 2
```

```
>>> print(n) 2
>>> type(n)
```

```
<class 'str'>
```

```
>>> f = input("digite um número real: ") digite um número real: 4.83
>>> print(f) 4.83
```



```
>>> type(f)
```

```
<class 'str'>
```

```
>>>
```

Nesse exemplo, o primeiro comando `input` apresenta a mensagem “Digite algo:”, indicando ao usuário o que deve ser feito. Qualquer coisa pode ser digitada e, após pressionar *Enter*, o objeto `x` é carregado com o que quer que tenha sido digitado. A leitura sempre resulta em uma cadeia de texto carregada no objeto de destino. Se forem digitados apenas algarismos, ainda assim a leitura resultará em uma cadeia de caracteres. Isso pode ser constatado por meio dos objetos `n` e `f`, nos quais, aparentemente, foram digitados o que seriam, respectivamente, um número inteiro e um real.

Porém, ao utilizar o comando `type`, verifica-se que ambos são do tipo `string` (`str`).

Em resumo, o comando `input` retorna exclusivamente cadeias de caracteres. Como fazer, então, caso se necessite ler números inteiros ou reais? A resposta para isso são as funções de conversão de tipo.

Funções de conversão entre tipos simples

Estas funções permitem realizar a conversão entre tipos de dados simples, conforme indicado no Quadro.

Função	Descrição
<code>str</code> (argumento)	Converte o argumento para cadeia de texto.
<code>int</code> (argumento)	Converte o argumento para um número inteiro, se for possível. Caso não seja possível, gera um erro.
<code>float</code> (argumento)	Converte o argumento para um número real, se for possível. Caso não seja possível, gera um erro.

Quadro 2.4 Funções de conversão de tipo.

O Exemplo 2.10 mostra diversos casos de conversão utilizando essas



funções.

Exemplo 2.10 Uso das funções de conversão de tipo

```
>>> x = '19'
```

```
>>> type(x)
```

```
<class 'str'>
```

```
>>> a = int(x)
```

```
>>> type(a)
```

```
<class 'int'>
```

```
>>> x = '3.75'
```

```
>>> type(x)
```

```
<class 'str'>
```

```
>>> r = float(x)
```

```
>>> type(r)
```

```
<class 'float'>
```

```
>>> b = a + r
```

```
>>> print(b) 22.75
```

```
>>> x = str(b)
```

```
>>> print(x) 22.75
```




```
>>> type(x)
```

```
<class 'str'>
```

```
>>>
```

Unir essas funções com o comando `input` é uma possibilidade utilizada em Python para a leitura de objetos com conteúdo numérico, seja inteiro ou real, da seguinte maneira:

```
N = int(input("Digite um número inteiro")) ou
```

```
F = float(input("Digite um número inteiro"))
```

Comentários no código

A inserção de comentários no código do programa é uma prática normal. Em função disso, toda linguagem de programação tem alguma maneira de permitir que comentários sejam inseridos nos programas. O objetivo é adicionar descrições em partes do código, seja para documentá-lo ou para adicionar uma descrição do algoritmo implementado. Os programadores também os utilizam para marcar que determinada linha, ou um conjunto de linhas, não devem ser processadas pelo interpretador, sem precisar excluí-las.

Em Python, existem duas maneiras de inserir comentários.

1. **Opção para uma linha:** a primeira forma usa o caractere `#` para comentar uma única linha. Não necessariamente esse caractere precisa ser posicionado no início da linha. Quando esse caractere é utilizado, o interpretador ignorará todo o restante da linha até o seu final. Veja a seguir:

```
# Esta linha inteira é um comentário e o interpretador a ignora X = 25 # Daqui  
para a frente é comentário  
print(X)
```

2. **Opção para múltiplas linhas:** a segunda forma possível utiliza três aspas duplas para abrir o bloco de comentário com muitas linhas e outras três aspas



duplas para fechar o bloco. É possível obter o mesmo resultado colocando aspas simples no lugar das aspas duplas. Essa construção não é exatamente um bloco de comentário. É algo a mais, conhecido como *docstrings*, quando utilizado, por exemplo, dentro de funções (veja o Capítulo 5). Os *docstrings* devem acompanhar a indentação (isso é visto no Capítulo 3) do código. Os *docstrings* não são empregados pelo interpretador para gerar qualquer código executável, e é por esse motivo que são utilizados como comentários.

```
"""
```

Tudo o que estiver entre as três aspas não vai gerar código pelo interpretador

```
"""
```

```
'''
```

Podem ser aspas simples também '''

