1.8. Primeiros Passos com Dados

Nós afirmamos acima que o Python suporta o paradigma de programação orientada a objetos. Isso significa que o Python considera os dados como o ponto focal do processo de resolução de problemas. Em Python, assim como em qualquer outra linguagem de programação orientada a objetos, definimos uma **classe** para ser uma descrição de como os dados se parecem (o estado) e o que os dados podem fazer (o comportamento). As classes são análogas aos tipos de dados abstratos porque um usuário de uma classe só vê o estado e o comportamento de um item de dados. Os itens de dados são chamados de **objetos** no paradigma orientado a objeto. Um objeto é uma instância de uma classe.

1.8.1. Tipos de Dados Atômicos Nativos

Vamos começar nossa revisão considerando os tipos de dados atômicos. O Python tem duas classes numéricas nativas principais que implementam os tipos inteiro e ponto flutuante. Essas classes do Python são chamadas de `int` e `float`. As operações aritméticas padrão, +, -, *, / e ** (exponenciação), pode ser usado com parênteses forçando a ordem de operações fora da precedência normal do operador. Outras operações muito úteis são o operador resto de divisão (módulo), %, e divisão inteira, //. Note que quando dois inteiros são divididos, o resultado é um ponto flutuante. O operador de divisão inteira retorna a porção inteira do quociente truncando qualquer parte fracionária.

Run | Load History | Show CodeLens

```
print(2+3*4)
print((2+3)*4)
print(2**10)
print(6/3)
print(7/3)
print(7//3)
print(7%3)
print(3/6)
print(3//6)
print(3%6)
print(2**100)

```

ActiveCode: 1 Operadores aritméticos básicos (intro_1)

O tipo de dado booleano, implementado como a classe `bool` do Python, será bastante útil para representar valores booleanos. Os possíveis valores de estado para um objeto booleano são `True` e `False` com os operadores booleanos, `and`, `or` e `not`.

saoPython.html)

```
>>> True
True
>>> False
False
>>> False or True
True
>>> not (False or True)
False
>>> True and True
True
```

Objetos de dados booleanos também são usados como resultado para operadores de comparação como igualdade (==) e maior que ($>$). Além disso, operadores relacionais e operadores lógicos podem ser combinados para formar questões lógicas complexas. A Tabela 1 mostra os operadores relacionais e lógicos com exemplos mostrados no trecho de código seguinte.

Tabela 1: Operadores Relacionais e Lógicos

Nome do Operador	Operador	Descrição
menor que	$<$	operador menor que
maior que	$>$	operador maior que
menor que ou igual	$<=$	operador menor que ou igual a
maior que ou igual	$>=$	operador maior que or igual a
igual	$==$	operador igualdade
não igual	$!=$	operador desigualdade
e lógico	and	Resulta em True quando ambos operandos são True
ou lógico	or	Resulta em True quando ao menos um dos operandos é True
não lógico	not	Nega o valor, False se torna True, True se torna False

Run Load History Show CodeLens

```
1 print(5==10)
2 print(10 > 5)
3 print((5 >= 1) and (5 <= 10))
4
```

saoPython.html)

ActiveCode: 2 Operadores Relacionais e Lógicos Básicos (intro_2)

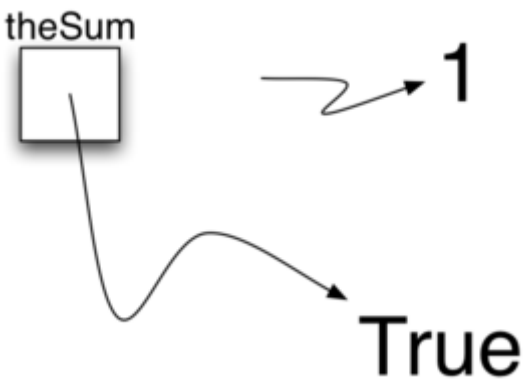
Identificadores são usados em linguagens de programação como nomes. Em Python, identificadores começam com uma letra ou um sublinhado (_), são sensíveis a letras minúsculas e maiúsculas, e podem ser de qualquer comprimento. Lembre-se que é sempre uma boa idéia usar nomes que transmitem significado para que o código de seu programa seja mais fácil de ler e entender.

Uma variável em Python é criada quando um nome é usado pela primeira vez do lado esquerdo de um comando de atribuição. Comandos de atribuição fornecem uma maneira de associar um nome a um valor. A variável manterá uma referência a um pedaço dos dados e não os dados em si. Considere o seguinte trecho de código:

```
>>> theSum = 0
>>> theSum
0
>>> theSum = theSum + 1
>>> theSum
1
>>> theSum = True
>>> theSum
True
```

O comando de atribuição `theSum = 0` cria uma variável chamada `theSum` e permite manter a referência ao objeto de dados `0` (veja Figura 3). Em geral, o lado direito do comando de atribuição é avaliado e uma referência ao objeto de dados resultante é “atribuída” ao nome no lado esquerdo. Neste ponto em nosso exemplo, o tipo da variável é inteiro por ser esse o tipo de dado atualmente sendo referido por `theSum`. Se o tipo de dado for modificado (veja Figura 4), como mostrado acima com o valor booleano `True`, o mesmo acontece com o tipo da variável (`theSum` é agora do tipo booleano). O comando de atribuição altera a referência sendo mantida pela variável. Essa é uma característica dinâmica do Python. A mesma variável pode se referir a muitos tipos diferentes de dados.

Figure 3: Variáveis mantém referências a objetos de dados



saoPython.html)

Figure 4: Atribuição modifica a referência

1.8.2. Tipos de Dados Coletivos Nativos

Além das classes numéricas e booleanas, o Python possui várias classes nativas muito poderosas para coleções. Listas (tipo list), strings (tipo str) e tuplas (tipo tuple) são coleções ordenadas que são muito semelhantes na estrutura geral, mas tem diferenças específicas que devem ser entendidas para serem usadas devidamente. Conjuntos (tipo set) e dicionários (tipo dict) são coleções não ordenadas.

Uma **lista** é uma coleção ordenada de zero ou mais referências a objetos de dados em Python. As listas são escritas como valores separados por vírgulas e delimitadas por colchetes. A lista vazia é simplesmente `[]`. Listas são heterogêneas, o que significa que os objetos de dados não precisam ser todos da mesma classe e a coleção pode ser atribuída a uma variável como abaixo. O trecho de código a seguir mostra vários objetos de dados do Python em uma lista.

```
>>> [1,3,True,6.5]
[1, 3, True, 6.5]
>>> minhaLista = [1,3,True,6.5]
>>> minhaLista
[1, 3, True, 6.5]
```

Observe que, quando o Python avalia uma lista, a própria lista é retornada. No entanto, para lembrar da lista para processamento posterior, sua referência precisa ser atribuída a uma variável.

Como as listas são consideradas sequencialmente ordenadas, elas suportam um número de operações que podem ser aplicadas a qualquer sequência do Python. A Tabela 2 revisa essas operações e o trecho de código seguinte fornece exemplos de seu uso.

Tabela 2: Operações sobre Qualquer Sequência em Python

Nome da Operação	Operador	Descrição
indexação	[ ]	Acessa um elemento da sequência
concatenação	+	Combina duas sequências
repetição	*	Concatena um certo número de vezes
pertinência	in	responde se um item pertence à sequência
comprimento	len	fornece o número de itens da sequência
fatiamento	[:]	Extrai parte de uma sequência

Note que os índices das listas (sequências) começam a contar do 0. A operação de fatiamento, minhaLista[1:3], retorna uma lista de itens que começam com o item indexado por 1 até, mas não incluindo o item indexado por 3.

saoPython.html)

As vezes, você desejará inicializar uma lista. Isso pode ser realizado rapidamente usando repetição. Por exemplo,

```
>>> minhaLista = [0] * 6
>>> minhaLista
[0, 0, 0, 0, 0, 0]
```

Um lado muito importante relacionado ao operador de repetição é que o resultado é uma repetição de referências aos objetos de dados na sequência. Isso pode ser melhor visto considerando o seguinte trecho de código:

Run Load History Show CodeLens

```
1 minhaLista = [1,2,3,4]
2 A = [minhaLista]*3
3 print(A)
4 minhaLista[2]=45
5 print(A)
6
```

ActiveCode: 3 Repetição de Referências (intro_3)

A variável `A` contém uma coleção de três referências à lista original chamada `minhaLista`. Observe que uma alteração em um elemento de `minhaLista` aparece em todas as três ocorrências em `A`.

As listas suportam vários métodos que serão usados para criar estruturas de dados. A Tabela 3 fornece um resumo. Exemplos são mostrados em seguida.

Tabela 3: Métodos Proporcionados por Listas em Python

Nome do Método	Exemplo de Uso	Descrição
append	`lista.append(item)`	Adiciona um novo item ao final de uma lista
insert	`lista.insert(i,item)`	Insere um item na i-ésima posição de uma lista
pop	`lista.pop()`	Remove e retorna o último item de uma lista
pop	`lista.pop(i)`	Remove e retorna o i-ésimo item de uma lista
sort	`lista.sort()`	Modifica uma lista para ficar ordenada
reverse	`lista.reverse()`	Modifica uma lista, invertendo a ordem dos itens
del	`del lista[i]`	Exclui o item na posição i
index	`lista.index(item)`	Retorna o índice da primeira ocorrência de `item`
count	`lista.count(item)`	Retorna o número de ocorrências de `item`

saoPython.html)

Nome do Método	Exemplo de Uso	Descrição
`remove`	`lista.remove(item)`	Remove a primeira ocorrência de `item`

Run Load History Show CodeLens

```
minhaLista = [1024, 3, True, 6.5]
minhaLista.append(False)
print(minhaLista)
minhaLista.insert(2,4.5)
print(minhaLista)
print(minhaLista.pop())
print(minhaLista)
print(minhaLista.pop(1))
print(minhaLista)
minhaLista.pop(2)
print(minhaLista)
minhaLista.sort()
print(minhaLista)
minhaLista.reverse()
print(minhaLista)
print(minhaLista.count(6.5))
print(minhaLista.index(4.5))
minhaLista.remove(6.5)
print(minhaLista)
del minhaLista[0]
print(minhaLista)

```

ActiveCode: 4 Exemplos de Métodos de Lista (intro_5)

Você pode ver que alguns dos métodos, como `pop`, retornam um valor e também modificam a lista. Outros, como `reverse`, simplesmente modificam a lista sem valor de retorno. `pop` por default remove e devolve o último item da lista mas também pode remover e devolver um item específico. O intervalo de índices começando de 0 é novamente usado por esses métodos. Você também deve notar a notação familiar usando “ponto” para pedir a um objeto para invocar um método. `minhaLista.append(False)` pode ser lido como “pergunte ao objeto `minhaLista` para executar seu método `append` passando-lhe o valor `False`”. Mesmo objetos de dados simples, como inteiros, podem invocar métodos dessa maneira.

```
>>> (54).__add__(21)
75
>>>
```

saoPython.html)

Neste fragmento estamos pedindo ao objeto inteiro `54` para executar seu método `add` (chamado `__add__` em Python) e passando-lhe `21` como valor a adicionar. O resultado é a soma, `75`. Claro, nós geralmente escrevemos isto como `54 + 21`. Falaremos muito mais sobre esses métodos mais tarde nesta seção.

Uma função comum do Python que é frequentemente discutida em conjunto com listas é a função `range`. A função `range` produz um **objeto de intervalo** (*range object*) que representa uma sequência de valores. Usando a função `list`, é possível ver o valor do objeto de intervalo como uma lista. Isto é ilustrado abaixo.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
range(5, 10)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

O objeto de intervalo representa uma sequência de inteiros. Por padrão, ela começa com 0. Se você fornecer mais parâmetros, ela começará e terminará em pontos específicos e pode até mesmo pular itens. Em nosso primeiro exemplo, `range(10)`, a sequência começa com 0 e vai até, mas não inclui, o 10. Em nosso segundo exemplo, `range(5,10)` começa em 5 e vai até, mas não inclui, o 10. `range(5,10,2)` tem efeito similar, mas pula de dois em dois (mais uma vez, 10 não está incluído).

Strings são coleções sequenciais de zero ou mais letras, números e outros símbolos. Nós chamamos essas letras, números e outros símbolos de *caracteres*. Valores literais de string são diferenciados dos identificadores usando aspas (simples ou duplas).

```
>>> "David"
'David'
>>> meuNome = "David"
>>> meuNome[3]
'i'
>>> meuNome*2
'DavidDavid'
>>> len(meuNome)
5
>>>
```

Como strings são sequências, todas as operações de sequência descritas acima funcionam como seria de se esperar. Além disso, as strings têm vários métodos, alguns dos quais são mostrados na Tabela 4. Por exemplo,

saoPython.html)

```
>>> meuNome
'Daniel'
>>> meuNome.upper()
'DANIEL'
>>> meuNome.center(10)
'  Daniel  '
>>> meuNome.find('n')
2
>>> meuNome.split('n')
['Da', 'iel']
```

Destes, `split` será muito útil para processar dados. `split` pegará uma string e retornará uma lista de strings usando a string de entrada como um ponto de divisão. No exemplo, `v` é o ponto de divisão. Se nenhuma divisão for especificada, o método de divisão procurará caracteres de espaço em branco, como tabulação, nova linha e espaço.

Tabela 4: Métodos Nativos do Python para Strings

Nome do método	Exemplo de Uso	Descrição
`center`	`umastring.center(w)`	Retorna uma string centrada em um campo de tamanho `w`
`count`	`umastring.count(item)`	Retorna o número de ocorrências de `item` na string
`ljust`	`umastring.ljust(w)`	Retorna uma string justificada à esquerda em um campo de tamanho `w`
`lower`	`umastring.lower()`	Retorna uma string em minúsculas
`rjust`	`umastring.rjust(w)`	Retorna uma string justificada à direita em um campo de tamanho `w`
`find`	`umastring.find(item)`	Retorna o índice da primeira ocorrência de `item`
`split`	`umastring.split(schar)`	Divide uma string em substrings em `schar`

Uma diferença importante entre listas e strings é que as listas podem ser modificado enquanto strings não podem. Isso é chamado de **mutabilidade**. As listas são mutáveis; strings são imutáveis. Por exemplo, você pode alterar um item em uma lista usando indexação e atribuição. Com uma string essa mudança não é permitida.

saoPython.html)

```
>>> minhaLista
[1, 3, True, 6.5]
>>> minhaLista[0]=2**10
>>> minhaLista
[1024, 3, True, 6.5]
>>>
>>> meuNome
'daniel'
>>> meuNome[0]='X'

Traceback (most recent call last):
  File "<pyshell#84>", line 1, in -toplevel-
    meuNome[0]='X'
TypeError: object doesn't support item assignment
>>>
```

As tuplas são muito semelhantes às listas por serem sequências de dados heterogêneas. A diferença é que uma tupla é imutável, como uma string. Uma tupla não pode ser alterada. As tuplas são escritas como valores delimitados por vírgula fechados entre parênteses. Como sequências, eles podem usar qualquer operação descrita acima. Por exemplo,

```
>>> minhaTupla = (2,True,4.96)
>>> minhaTupla
(2, True, 4.96)
>>> len(minhaTupla)
3
>>> minhaTupla[0]
2
>>> minhaTupla * 3
(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)
>>> minhaTupla[0:2]
(2, True)
>>>
```

Porém, se você tentar modificar um item em uma tupla, você vai obter um erro. Note que a mensagem de erro fornece o local e a razão do problema.

```
>>> minhaTupla[1]=False

Traceback (most recent call last):
  File "<pyshell#137>", line 1, in -toplevel-
    minhaTupla[1]=False
TypeError: object doesn't support item assignment
>>>
```

Um conjunto é uma coleção não ordenada de zero ou mais objetos de dados imutáveis do Python. Conjuntos não permitem duplicatas e são gravados como valores delimitados por vírgula fechados entre chaves. O conjunto vazio é representado por `set()`. Conjuntos são heterogêneos e a coleção pode ser atribuída a uma variável como abaixo.

saoPython.html)

```
>>> {3,6,"gato",4.5,False}
{False, 4.5, 3, 6, 'gato'}
>>> umConjunto = {3,6,"gato",4.5,False}
>>> umConjunto
{False, 4.5, 3, 6, 'gato'}
>>>
```

Mesmo que os conjuntos não sejam considerados sequenciais, eles suportam algumas das operações familiares apresentadas anteriormente. A Tabela 5 descreve estas operações e o trecho de código seguinte dá exemplos de seu uso.

Tabela 5: Operações sobre Conjuntos em Python

Nome do Operador	Operador	Descrição
pertinência	in	se pertence ao conjunto
length	len	Retorna a cardinalidade do conjunto
|	`conj1 | conj2`	Retorna um novo conjunto com todos os elementos de ambos os conjuntos
&	`conj1 & conj2`	Retorna um novo conjunto com apenas os elementos comuns a ambos
-	`conj1 - conj2`	Retorna um novo conjunto com todos os itens do primeiro conjunto, não no segundo
<=	`conj1 <= conj2`	Pergunta se todos os elementos do primeiro conjunto estão no segundo

```
>>> umConjunto
{False, 4.5, 3, 6, 'gato'}
>>> len(umConjunto)
5
>>> False in umConjunto
True
>>> "rato" in umConjunto
False
>>>
```

Os conjuntos suportam vários métodos que devem ser familiares àqueles que já trabalharam com eles em um contexto matemático. A Tabela 6 fornece um resumo e exemplos de seu uso. Note que `union`, `intersection`, `issubset` e `difference` possuem operadores que pode ser usados também.

saoPython.html)

Tabela 6: Métodos Fornecidos para Conjuntos em Python

Nome do Método	Exemplo de Uso	Descrição

Nome do Método	Exemplo de Uso	Descrição
`union`	`conj.union(otherset)`	Retorna um novo conjunto com todos os elementos de ambos os conjuntos
`intersection`	`conj.intersection(otherset)`	Retorna um novo conjunto com apenas os elementos comuns a ambos os conjuntos
`difference`	`conj.difference(otherset)`	Retorna um novo conjunto com todos os itens do primeiro conjunto, não no segundo
`issubset`	`conj.issubset(otherset)`	Pergunta se todos os elementos de um conjunto estão no outro
`add`	`conj.add(item)`	Adiciona item ao conjunto
`remove`	`conj.remove(item)`	Remove item do conjunto
`pop`	`conj.pop()`	Remove um elemento arbitrário do conjunto
`clear`	`conj.clear()`	Remove todos os elementos do conjunto

saoPython.html)

```
>>> umConjunto
{False, 4.5, 3, 6, 'gato'}
>>> outroConjunto = {99,3,100}
>>> umConjunto.union(outroConjunto)
{False, 4.5, 3, 100, 6, 'gato', 99}
>>> umConjunto | outroConjunto
{False, 4.5, 3, 100, 6, 'gato', 99}
>>> umConjunto.intersection(outroConjunto)
{3}
>>> umConjunto & outroConjunto
{3}
>>> umConjunto.difference(outroConjunto)
{False, 4.5, 6, 'gato'}
>>> umConjunto - outroConjunto
{False, 4.5, 6, 'gato'}
>>> {3,100}.issubset(outroConjunto)
True
>>> {3,100}<=outroConjunto
True
>>> umConjunto.add("casa")
>>> umConjunto
{False, 4.5, 3, 6, 'casa', 'gato'}
>>> umConjunto.remove(4.5)
>>> umConjunto
{False, 3, 6, 'casa', 'gato'}
>>> umConjunto.pop()
False
>>> umConjunto
{3, 6, 'casa', 'gato'}
>>> umConjunto.clear()
>>> umConjunto
set()
>>>
```

Nossa última coleção do Python é uma estrutura desordenada chamada **dicionário**. Dicionários são coleções de pares associados de itens onde cada par consiste de uma chave e um valor. Este par chave-valor é tipicamente escrito como chave:valor. Dicionários são escritos como pares chave:valor separados por vírgula e delimitados por chaves. Por exemplo,

```
>>> capitais = {'Amazonas':'Manaus','Paraná':'Curitiba'}
>>> capitais
{'Amazonas': 'Manaus', 'Paraná': 'Curitiba'}
>>>
```

Podemos manipular um dicionário acessando um valor por meio de sua chave ou adicionando outro par chave-valor. A sintaxe de acesso se parece muito com um acesso de sequência, exceto que em vez de usar o índice do item nós usamos o valor da chave. Para adicionar um novo valor é semelhante.

saoPython.html)

Run Load History Show CodeLens

```
1 capitais = {'Amazonas':'Manaus','Paraná':'Curitiba'}
2 print(capitais['Amazonas'])
3 capitais['Minas Gerais']='Belo Horizonte'
4 print(capitais)
5 capitais['Bahia']='Salvador'
6 print(len(capitais))
7 for k in capitais:
8    print(capitais[k]," é a capital de[a/o] ", k)
9
```

ActiveCode: 5 Usando um Dicionário (intro_7)

É importante notar que o dicionário é mantido sem nenhuma ordem em particular com relação às chaves. Assim, apesar do primeiro par adicionado tenha sido `'Amazonas': 'Manaus'` e o último `'Bahia': 'Salvador'`, ao percorrer o dicionário usando o comando `for` por exemplo (linha 7), a ordem das chaves é incerta. A ordem de uma chave depende da ideia de “hashing” que será explicada em mais detalhes no Capítulo 4. Também mostramos a função length executando o mesmo papel que nas coleções anteriores.

Os dicionários têm métodos e operadores, descritos na Tabela 7 e na Tabela 8. A sessão de Python os mostra em ação. Os métodos `keys`, `values` e `items` retornam objetos que contém os valores de interesse. Você pode usar a função `list` para convertê-los em listas. Observe também que existem duas variações no método `get`. Se a chave não estiver presente no dicionário, `get` retornará `None`. No entanto, um segundo parâmetro opcional pode especificar um outro valor de retorno.

Tabela 7: Operadores Fornecidos por Dicionários em Python

Operador	Exemplo	Descrição
`[]`	`myDict[k]`	Retorna o valor associado a `k`, caso contrário é um erro
`in`	`key in dicio`	Retorna `True` se key estiver no dicionário, `False` caso contrário
`del`	`del dicio[key]`	Remove o item do dicionário associado a key

saoPython.html)

```
>>> ramal={'daniel':1410,'adriana':1137}
>>> ramal
{'adriana': 1137, 'daniel': 1410}
>>> ramal.keys()
dict_keys(['adriana', 'daniel'])
>>> list(ramal.keys())
['adriana', 'daniel']
>>> ramal.values()
dict_values([1137, 1410])
>>> list(ramal.values())
[1137, 1410]
>>> ramal.items()
dict_items([('adriana', 1137), ('daniel', 1410)])
>>> list(ramal.items())
[('adriana', 1137), ('daniel', 1410)]
>>> ramal.get("pedro")
>>> ramal.get("pedro","SEM RAMAL")
'SEM RAMAL'
>>>
```

Table 8: Methods Provided by Dictionaries in Python

Nome do Método	Exemplo de Uso	Descrição
`keys`	`dicio.keys()`	Retorna as chaves do dicionário em um objeto dict_keys
`values`	`dicio.values()`	Retorna os valores do dicionário em um objeto dict_values
`items`	`dicio.items()`	Retorna os pares de chave-valor em um objeto dict_items
`get`	`dicio.get(k)`	Retorna o valor associado a `k`, `None` caso contrário
`get`	`dicio.get(k,alt)`	Retorna o valor associado a `k`, `alt` caso contrário

Rascunho

Essa área de trabalho é fornecida para sua conveniência. Você pode usar essa janela de activecode para testar qualquer coisa que desejar.

Run Load History Show CodeLens

```
1
2
3
```

saoPython.html)

ActiveCode: 6 (scratch_01_01)

(07-revisaoPython.html)

(09-entradaSaida.html)

 | Back to top

saoPython.html)