Capítulo 6

Strings

6.1 String é uma sequência

String é uma sequência de caracteres. Você pode acessar os caracteres um de cada vez com o operador colchetes:

```
>>> fruta = 'banana'
>>> letra = fruta[1]
```

A segunda instrução extrai o caractere na posição indexada 1 da variável `fruta` e atribui este valor à variável `letra`.

A expressão em colchetes é chamada de *index*. O index indica qual caractere da sequência você quer(daí o nome).

Mas você talvez não receba o que espera:

```
>>> print(letra)
a
```

Para a maior parte das pessoas, a primeira letra de "banana" é "b", não "a". Mas em Python, o index é um deslocamento desde o início da string, e o deslocamento da primeira letra é zero.

```
>>> letra = fruta[0]
>>> print(letra)
b
```

Então "b" é a 0ª letra de "banana", "a" é a 1ª letra e "n" é a 2ª letra.

Você pode utilizar qualquer expressão como index, incluindo variáveis e operadores, mas o valor deverá ser um inteiro. Caso contrário você tem:

```
>>> letra = fruta[1.5]
TypeError: string indices must be integers
```

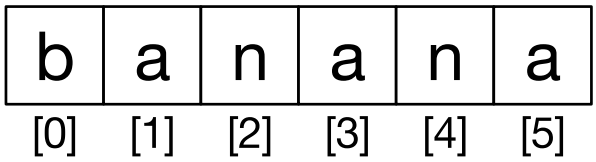

Figure 6.1: String Indexes

6.2 Obtendo o comprimento de uma string utilizando `len`

`len` é uma função interna que retorna o número de caracteres de uma string:

```
>>> fruta = 'banana'
>>> len(fruta)
6
```

Para obter a última letra de uma string, você pode ficar tentado a fazer algo desta natureza:

```
>>> comprimento = len(fruta)
>>> ultima = fruta[comprimento]
IndexError: string index out of range
```

A razão pelo `IndexError` é que não há uma letra em "banana" com o index 6. Desde que começamos a contar de zero, as seis letras estão enumeradas de 0 a 5. Para obter o último caractere, você precisa subtrair 1 de `comprimento`:

```
>>> ultima = fruta[comprimento-1]
>>> print(ultima)
a
```

Alternativamente, você pode utilizar de índices negativos, que faz a contagem ao contrário (de trás para frente). A expressão `fruta[-1]` retorna a última letra, `fruta[-2]` retorna a penúltima, e assim por diante.

6.3 Travessia de strings usando laço

Muito em computação envolve o processamento de strings, caractere por caractere. Geralmente começando pelo primeiro caractere da string, seleciona-se um por vez, alguma operação é realizada sobre ele e o processo é repetido até o fim da string. Esse modelo de processamento é chamado de *travessia (traversal)*. Uma forma de escrever uma travessia é através de um laço `while`.

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print(letra)
    indice = indice + 1
```

Este laço atravessa a string e imprime cada letra em uma linha diferente. A condição do laço é `indice < len(fruta)`, então quando `indice` é igual ao tamanho da string, a condição é falsa, e o corpo do laço não é executado. O último caractere acessado é o que possui o índice `len(fruit)-1`, que é o último caractere da string.

Exercício 1: Escreva um loop `while` que inicia no último caractere da string e caminha para o primeiro caractere, imprimindo cada letra em uma linha separada.

Outra forma de escrever uma travessia é com um laço `for`:

```
for char in fruta:
    print(char)
```

Cada vez que passa pelo laço, o próximo caractere da string é atribuído à variável `char`. O laço continua até o fim dos caracteres.

6.4 Segmentos de strings

Um segmento de uma string é chamado *slice*. A seleção de um slice é similar a seleção de um caractere:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

O operador retorna a parte da string que está entre os dois números, incluindo o primeiro número e excluindo o último.

Se você omitir o primeiro índice, o slice irá começar do início da string. Já se você omitir o segundo índice, o slice irá terminar no último caractere da string:

```
>>> fruta = 'banana'
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

Se o primeiro índice foi maior ou igual ao segundo índice, o resultado será uma *string vazia*, representada por duas aspas:

```
>>> fruta = 'banana'
>>> fruta[3:3]
''
```

Uma string vazia não contém nenhum caractere e possui comprimento 0, mas fora isso, é como qualquer outra string.

Exercício 2: Dado que `fruta` é uma string, qual o resultado de `fruta[:]`?

6.5 Strings são imutáveis

É tentador usar o operador do lado esquerdo de uma sentença com a intenção de mudar um caractere de uma string. Por exemplo:

```
>>> Saudacao = 'Alô, Mundo!'
>>> Saudacao[0] = 'O'
TypeError: 'str' object does not support item assignment
    (objeto 'str' não permite atribuição de item)
```

O "objeto" nesse caso é a string e o "item" é o caractere que você tentou atribuir. Por agora, um *objeto* é a mesma coisa que um valor, mas nós refinaremos essa definição mais tarde. Um *item* é um dos valores em uma sequência.

A razão para o erro é que strings são *imutáveis*, o que significa que você não pode alterar uma string já existente. O melhor a se fazer é criar uma nova string que é uma variação da original:

```
>>> saudacao = 'Alô, Mundo!'
>>> nova_saudacao = 'O' + saudacao[1:]
>>> print(nova_saudacao)
Olô, Mundo!
```

Esse exemplo concatena uma nova primeira letra com um segmento de `saudacao`. Isso não afeta a string original.

6.6 Laços e contagem

O programa a seguir conta o número de vezes que a letra "a" aparece em uma string:

```
palavra = 'banana'
contagem = 0
for letra in palavra:
    if letra == 'a':
        contagem = contagem + 1
print(contagem)
```

Esse programa demonstra outro padrão da computação chamado *contador*. A variável `contagem` é inicializada com 0 e então incrementada a cada vez que um "a" é encontrado. Quando o laço acaba, `contagem` tem como resultado o número total de a's.

Exercício 3: Encapsule esse código em uma função chamada `contagem`, e generalize para que ela aceite a string e a letra como argumentos.

6.7 O operador in

A palavra `in` é uma operador booleano que usa duas strings e retorna `True` se a primeira aparecer como uma substring na segunda:

```
>>> 'a' in 'banana'
True
>>> 'semente' in 'banana'
False
```

6.8 Comparação de strings

Os operadores de comparação funcionam em strings. Para verificar se duas delas são iguais:

```
if palavra == 'banana':
    print('Certo, bananas.')
```

Outras operações de comparação são úteis para organizar palavras em ordem alfabética:

```
if palavra < 'banana':
    print('Sua palavra,' + palavra + ', vem antes de banana.')
elif palavra > 'banana':
    print('Sua palavra,' + palavra + ', vem depois de banana.')
else:
    print('Certo, bananas.')
```

O Python não manipula letras maiúsculas e minúsculas do mesmo modo que as pessoas. Para ele, todas as letras maiúsculas vêm antes de todas as letras minúsculas, sendo assim:

```
Sua palavra, Banana, vem antes de banana.
```

Uma maneira comum de resolver esse problema é converter as sequências de caracteres em um formato padrão (por exemplo, todas minúsculas) antes de executar a comparação. Tenha isso em mente caso você tenha que se defender contra um homem armado com uma Banana.

6.9 Métodos da String

Strings são exemplos de *objetos* no Python. Um objeto contém tanto a informação (a própria string), como *métodos*, que são funções eficientes construídas dentro do objeto e disponíveis em qualquer *instância* do mesmo.

Python possui uma função chamada `dir`, que lista os métodos disponíveis em um objeto. A função `type` mostra o tipo do objeto e a função `dir` mostra os métodos disponíveis.

```
>>> coisa = 'Olá Mundo'
>>> type(coisa)
<class 'str'>
>>> dir(coisa)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
   (Retorna uma versão capitalizada de S, isto é, torna o primeiro
    caractere maiúsculo e o restante minúsculo.)
>>>
```

Você pode usar o comando `help` para ter acesso a uma simples documentação sobre o método. Uma melhor fonte de documentação para os métodos da string pode ser encontrada em: https://docs.python.org/library/stdtypes.html#string-methods.

Chamar um `método` é similar a chamar uma função (recebe argumentos e retorna um valor), mas a sintaxe é diferente. Chamamos um método anexando o nome dele ao nome da variável usando o ponto como delimitador.

Por exemplo, o método `upper` recebe uma string e retorna uma nova com todas as letras em maiúsculo:

Em vez da sintaxe de função `upper(palavra)`, é usado a sintaxe de método `palavra.upper()`.

```
>>> palavra = 'banana'
>>> nova_palavra = palavra.upper()
>>> print(nova_palavra)
BANANA
```

Essa notação, utilizando o ponto, especifica o nome do método, `upper`, e o nome da string em que o método está sendo utilizado, `palavra`. Os parênteses vazios indicam que o método não recebe argumentos.

A chamada de um método é denotada por *invocação*. Nesse caso, nós devemos dizer que estamos invocando `upper` em `palavra`.

Por exemplo, existe um método na string chamado `find` que procura pela posição de uma string dentro de outra:

```
>>> palavra = 'banana'
>>> indice = palavra.find('a')
>>> print(indice)
1
```

Nesse exemplo, invocamos `find` em `palavra` e passamos a letra que estamos procurando como argumento.

O método `find` pode encontrar substrings, assim como caracteres:

```
>>> palavra.find('na')
2
```

O método pode receber como segundo argumento o índice onde deve começar:

```
>>> palavra.find('na', 3)
4
```

Uma prática comum consiste em remover espaços em branco (espaços, tabs ou quebra de linha) do começo e do final da string, usando o método `strip`:

```
>>> linha = '  Aqui vamos nós  '
>>> linha.strip()
'Aqui vamos nós'
```

Alguns métodos como *startswith* retornam valores booleanos.

```
>>> linha = 'Tenha um bom dia'
>>> linha.startswith('Tenha')
True
>>> linha.startswith('t')
False
```

Você vai notar que `startswith` requer que ambos os caracteres comparados estejam em maiúsculo ou em minúsculo para combinar. Então, às vezes, nós pegamos uma linha e a mapeamos toda em caixa baixa, usando o método `lower`, antes de fazer qualquer checagem.

```
>>> linha = 'Tenha um bom dia'
>>> linha.startswith('t')
False
>>> linha.lower()
'tenha um bom dia'
>>> linha.lower().startswith('t')
True
```

Nesse último exemplo, o método `lower` é chamado e então nós usamos `startswith` para ver se a versão em letras minúsculas da string começa com a letra "t". Enquanto tomarmos cuidado com a ordem, nós podemos chamar múltiplos métodos em uma simples expressão.

**Exercício 4: Existe um método na string chamado `count` que é similar à função usada no exercício anterior. Leia a documentação desse método em:

https://docs.python.org/library/stdtypes.html#string-methods

Escreva uma invocação que conta o número de vezes que a letra "a" aparece em "banana".**

6.10 Particionando strings

Frequentemente, nós precisamos analisar o que há dentro de uma string e encontrar uma substring. Por exemplo, se nos for apresentada uma sequência de linhas formatadas da seguinte forma:

`From stephen.marquard@`*`uct.ac.za`*`Sat Jan  5 09:14:16 2008`

e nós quisermos particionar somente a segunda metade do endereço (i.e.,`uct.ac.za`) de cada linha, podemos fazer isso utilizando o método `find` e o fatiamento de strings.

Primeiramente, acharemos a posição do sinal arroba (@) na string. Depois, acharemos a posição do primeiro espaço *após* o arroba. Só então utilizaremos o fatiamento para extrair a parte da string que estamos buscando.

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print(atpos)
21
>>> sppos = data.find(' ',atpos)
>>> print(sppos)
31
>>> host = data[atpos+1:sppos]
>>> print(host)
uct.ac.za
>>>
```

Utilizamos a versão do método `find` que nos permite especificar a posição na string onde queremos começar a busca. Quando fatiamos a string, extraímos os caracteres de "um a frente do sinal arroba até o caractere espaço, *sem incluí-lo*".

A documentação para o método `find` está disponível em

https://docs.python.org/library/stdtypes.html#string-methods.

6.11 Operador de Formatação

O *operador de formatação* `%` nos permite construir strings, substituindo partes destas strings pela informação contida em variáveis. Quando aplicamos à inteiros,

% é o operador de módulo. Porém, quando o primeiro operando é uma string, % é o operador de formatação.

O primeiro operando é a *string de formatação*, que contém uma ou mais *sequências de formatação* que especificam como o segundo operando é formatado. O resultado é uma string.

Por exemplo, a sequência de formatação %d significa que o segundo operando deve ser formatado como um número inteiro ("d" significa "decimal"):

```
>>> camelos = 42
>>> '%d' % camelos
'42'
```

O resultado é a string '42', que não é para ser confundida com o valor inteiro 42.

Uma sequência de formatação pode aparecer em qualquer lugar dentro de uma string, te permitindo alocar um valor em uma frase:

```
>>> camelos = 42
>>> 'Eu vi %d camelos.' % camelos
'Eu vi 42 camelos.'
```

Se existe mais de uma sequência de formatação em uma string, o segundo argumento tem que ser uma tupla [1]. Cada uma das sequências é combinada com um elemento da tupla, em ordem.

O seguinte exemplo usa %d para formatar um inteiro, %g para formatar um número de ponto flutuante (não pergunte o porquê), e %s para formatar uma string:

```
>>> 'Em %d anos eu vi %g %s.' % (3, 0.1, 'camelos')
'Em 3 anos eu vi 0.1 camelos.'
```

O número de elementos na tupla deve ser igual ao número de sequências de formatação na string. Os tipos dos elementos também devem corresponder aos tipos das sequências:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
          (argumentos insuficientes para a string de formatação)
>>> '%d' % 'reais'
TypeError: %d format: a number is required, not str
          (formatação %d: um número é requerido, não uma str)
```

No primeiro exemplo, não há elementos suficientes; no segundo, o tipo do elemento é incorreto.

O operador de formatação é poderoso, mas pode ser difícil de se usar. Você pode ler mais sobre em:

https://docs.python.org/library/stdtypes.html#printf-style-string-formatting.

[1] Uma tupla é uma sequência de valores separados por vírgulas dentro de um par de parênteses. Nós vamos tratar sobre tuplas no Capítulo 10

6.12 Debugging

Uma habilidade que você deveria desenvolver ao programar é a de sempre se perguntar, "O que poderia dar errado aqui?" ou, de maneira semelhante, "Qual loucura o usuário poderia fazer para gerar um problema no nosso programa (aparentemente) perfeito?"

Por exemplo, observe o programa que utilizamos para demonstrar o laço `while` no capítulo de iterações:

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')

# Code: http://www.py4e.com/code3/copytildone2.py
```

Observe o que acontece quando o usuário introduz uma linha vazia na entrada:

```
> Olá
Olá
> # não mostre isto
> Mostre isto!
Mostre isto!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#':
IndexError: string index out of range
           (índice fora dos limites da string)
```

O código funciona bem até o aparecimento da linha vazia, pois não há o caractere da posição 0, o que resulta no traceback (erro). Existem duas soluções para transformar a terceira linha do código em uma linha "segura", mesmo que a entrada seja vazia.

Uma alternativa seria simplesmente utilizar o método `startswith`, que retorna `False` se a string estiver vazia.

```
if line.startswith('#'):
```

Outra forma seria escrever a instrução `if` utilizando um padrão *guardião*, para assegurar que a segunda expressão lógica seja testada apenas quando a string conter no mínimo 1 caractere:

```
if len(line) > 0 and line[0] == '#':
```

6.13 Glossário

contador Uma variável que é utilizada para contar algo. Geralmente é inicializada com zero e incrementada. fatia
Uma parte de uma string especificada por uma intervalo de índices.

flag Uma variável boleana utilizada para indicar quando uma condição é verdadeira ou falsa.

imutável A propriedade de uma sequência cujos os itens não podem ser atribuídos ou alterados.

índice Um valor inteiro utilizado para selecionar um item em uma sequência, como um caractere em uma string.

invocação Uma declaração que chama um método.

item Um dos valores em uma sequência.

método Uma função que é associada a um objeto e chamada utilizando a notação com um ponto. objeto
Algo ao qual uma variável pode se referir. Por agora, você pode utilizar "objeto" e "valor" indistintamente.

operador de formatação Um operador,%, que recebe uma string de formatação e uma tupla e gera uma string que inclui os elementos da tupla formatados conforme especificado pela string.

pesquisa Um padrão de travessia que para quando encontra o que estava procurando.

sequência Um conjunto ordenado; ou seja, um conjunto de valores em que cada elemento é identificado por um índice.

sequência de formatação Uma sequência de caracteres em uma string de formatação, como `%d`, que especificam como um valor deve ser formatado

string de formatação Uma string, utilizada com o operador de formatação, que contém sequências de formatação.

string vazia Uma string com nenhum caractere e de dimensão 0, representada por duas aspas.

travessia Iterar através dos elementos de uma sequência, realizando uma operação similar em cada um deles.

6.14 Exercícios

Exercício 5: Utilize o seguinte código em Python que guarda uma string:

`str = 'X-DSPAM-Confidence:0.8475'` **Use a função `find` e o fatiamento de strings para extrair a porção da string depois do sinal de dois pontos e use a função `float` para converter a string extraída em um número de ponto flutuante.**

Exercícios 6: Leia a documentação dos métodos da string em https://docs.python.org/library/stdtypes.html#string-methods Você pode querer experimentar alguns deles para ter certeza que você entendeu como eles funcionam. `strip` e `replace` são particularmente úteis.

A documentação utiliza uma sintaxe que pode ser confusa. Por exemplo, em `find(sub[, start[, end]])`, as chaves indicam argumentos opcionais,

então, `sub` é necessário, mas `start` é opcional, e se você incluir `start`, logo `end` é opcional.