The Mathematica® Journal

Strings
Chapter 9
Programming with Mathematica: An Introduction

Paul Wellin

This article is an excerpt from the recently released book, *Programming with Mathematica: An Introduction* by Paul Wellin © 2013 [1]. The book, which follows on the well-known *An Introduction to Mathematica Programming*, provides an example-driven primer on the foundations of the *Mathematica* programming language.

Strings are used across many disciplines to represent filenames, data, and other objects: linguists working with text data study representation, classification, and patterns involved in audio and text usage; biologists dealing with genomic data as strings are interested in sequence structure and assembly and perform extensive statistical analysis of their data; programmers operate on string data for such tasks as text search, file manipulation, and text processing. Strings are so ubiquitous that almost every modern programming language has a string datatype and dozens of functions for operating on and with strings.

■ Introduction

In *Mathematica*, strings are represented by any concatenation of characters enclosed in double quotes.

```
StringQ["The magic words are squeamish ossifrage."]
```

```
True
```

Strings are also used to represent file names that you import and export.

```
Import["ExampleData/ocelot.jpg"]
```

Strings are used as arguments, option values, and as the output to many functions.

```
GenomeData["SNORD107"]
```

```
GGTTCATGATGACACAGGACCTTGTCTGAACATAATGATTTCAAAATTTGAGCTTAAAA⋱
    ATGACACTCTGAAATC
```

```
StringQ[%]
```

```
True
```

In this chapter we will introduce the tools available for working with strings in *Mathematica*. We will begin with a look at the structure and syntax of strings, then move on to a discussion of the many high-level functions that are optimized for string manipulation. String patterns follow on the discussion of patterns in Chapter 4 and we will introduce an alternative syntax (regular expressions) that provides a very compact mechanism for working with strings. The chapter closes with several applied examples drawn from computer science (checksums) as well as bioinformatics (working with DNA sequences) and also word games (anagrams, blanagrams).

■ 9.1. Structure and Syntax

Strings are expressions consisting of a number of characters enclosed in quotes. The characters can be anything you can type from your keyboard, including uppercase and lowercase letters, numbers, punctuation marks, and spaces. For example, here is the standard set of printable Ascii characters.

```
CharacterRange[" ", "~"]
```

```
{ , !, ", #, $, %, &, ', (, ), *, +, ,, -, ., /, 0, 1, 2,
 3, 4, 5, 6, 7, 8, 9, :, ;, <, =, >, ?, @, A, B, C, D, E,
 F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X,
 Y, Z, [, \, ], ^, _, `, a, b, c, d, e, f, g, h, i, j, k,
 l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, {, |, }, ~}
```

Other character sets are available as well. For example, here are the lowercase Greek letters. These are typically entered from one of *Mathematica*'s many built-in character palettes, or using a keyboard shortcut such as ESC-a-ESC for α.

```
CharacterRange["α", "ω"]
```

```
{α, β, γ, δ, ε, ζ, η, θ, ι, κ, λ,
 μ, ν, ξ, ο, π, ρ, ς, σ, τ, υ, φ, χ, ψ, ω}
```

When *Mathematica* displays a string in output, it appears without the quotes. This is the default behavior of the formatting rules for `OutputForm`.

```
"The magic words are squeamish ossifrage."
```

```
The magic words are squeamish ossifrage.
```

Use `InputForm` or `FullForm` to display these quotes in output.

```
FullForm["The magic words are squeamish ossifrage."]
```

```
"The magic words are squeamish ossifrage."
```

Various predicates test whether a string consists entirely of letters, or uppercase and lowercase letters.

```
LetterQ["ossifrage"]
```

```
True
```

```
LetterQ["x1"]
```

```
False
```

```
LowerCaseQ["strings"]
```

```
True
```

Use === (`SameQ`) to test for equality of strings.

```
"sty" === "sty "
```

```
False
```

Several functions are available for working with the structure of strings.

```
Head["The magic words are squeamish ossifrage."]

String

StringLength["The magic words are squeamish ossifrage."]

40
```

`StringLength` also works with lists of strings. In other words, it has the `Listable` attribute.

```
StringLength[{"How", "I", "wish", "I", "could",
  "calculate", "pi"}]

{3, 1, 4, 1, 5, 9, 2}
```

□ Character Codes

One way to work with strings is to convert them to a list of character codes and then operate on the codes using mathematical functions. Each character in a computer's character set is assigned a number, called its *character code*. By general agreement, almost all computers use the same character codes, called the *Ascii code*. In this code, the uppercase letters *A*, *B*, …, *Z* are assigned the numbers 65, 66, …, 90 while the lowercase letters *a*, *b*, …, *z* have the numbers 97, 98, …, 122 (note that the number of an uppercase letter is 32 less than its lowercase version). The numbers 0, 1, …, 9 are coded as 48, 49, …, 57 while the punctuation marks period, comma, and exclamation point have the codes 46, 44, and 33, respectively. The space character is represented by the code 32. Table 9.1 shows the characters and their codes.

Characters	Ascii codes
A, B, …, Z	65, 66, …, 90
a, b, …, z	97, 98, …, 122
0, 1, …, 9	48, 49, …, 57
. (period)	46
, (comma)	44
? (question mark)	63
␣ (space)	32

▲ **Table 9.1.** Ascii character codes

Here are the printable Ascii characters.

```
FromCharacterCode[Range[32, 126]]
```

```
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ
[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

`ToCharacterCode[`*char*`]` converts any string character *char* to its Ascii code.

```
ToCharacterCode[%]
```

```
{32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104,
 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126}
```

You can also get a list of the characters in a range if you know how they are ordered by their character codes.

```
CharacterRange["a", "z"]
```

```
{a, b, c, d, e, f, g, h, i, j, k,
 l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

```
Flatten[ToCharacterCode[%]]
```

```
{97, 98, 99, 100, 101, 102, 103, 104,
 105, 106, 107, 108, 109, 110, 111, 112, 113,
 114, 115, 116, 117, 118, 119, 120, 121, 122}
```

Characters from other languages can also be used, for example, Greek and Japanese.

```
FromCharacterCode[Range[913, 1009]]
```

```
ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡ  ΣΤΥΦΧΨΩΪΫάέήίΰαβγδεζηθικλμνξοπρςστυφχψωϊϋ⋱
    όύώϐ  ϑϒϓϔϕϖϗϘϙϚϛϜϝϞϟϠϡϢϣϤϥϦϧϨϩϪϫϬϭϮϯϰϱ
```

```
FromCharacterCode[Range[30010, 30030]]
```

```
町画甼甽甾甿畀畁畂畃畄畅畆畇畈畉畊畋界畍畎
```

Unicode charts for many languages are available online (for example, www.unicode.org/charts). With these charts you can find the hexadecimal code for characters in many different languages. For Gujarati, the first character in its code table has hex value 0A90. Here we convert from base 16 and then display the character.

```
16^^0A90
```

```
2704
```

```
FromCharacterCode[%]
```

```
ઐ
```

Using the character code representation of characters, the following series of computations changes a word from lowercase to uppercase.

```
ToCharacterCode["mathematica"]
```

```
{109, 97, 116, 104, 101, 109, 97, 116, 105, 99, 97}
```

```
% - 32
```

```
{77, 65, 84, 72, 69, 77, 65, 84, 73, 67, 65}
```

```
FromCharacterCode[%]
```

```
MATHEMATICA
```

Or, simply use a built-in function that is designed specifically for this task.

```
ToUpperCase["mathematica"]
```

```
MATHEMATICA
```

□ Sorting Lists of Characters

As a practical example of the use of character codes, we will extend the simple sorting function from Chapter 4 to work with lists of string characters. Although written to operate on numbers, this rule can be overloaded to work on characters by making only a few small changes. Here is the original rule from Section 4.3.

```
listSort =
  {{x___, a_?NumericQ, b_?NumericQ, y___} :→
    {x, b, a, y} /; b < a};
```

The first change is to check that the patterns `a` and `b` have head `String` instead of testing for numbers with the predicate `NumericQ`. Second, instead of the numerical comparison `b < a`, we need to compare their character codes.

```
ToCharacterCode[{"q", "t"}]
```

```
{{113}, {116}}
```

```
charSort = {x___, a_String, b_String, y___} :>
  {x, b, a, y} /; First[ToCharacterCode[b]] <
    First[ToCharacterCode[a]]
```

```
{x___, a_String, b_String, y___} :> {x, b, a, y} /;
  First[ToCharacterCode[b]] < First[ToCharacterCode[a]]
```

Here is a list of characters.

```
chars = {"d", "h", "c", "m", "r", "l", "c", "h", "t", "d", "j"};
```

Here is the sort.

```
chars //. charSort
```

```
{c, c, d, d, h, h, j, l, m, r, t}
```

Section 9.5 explores the use of character codes to create hash tables, or checksums.

□ Ordered Words

When studying word or language structure, a common task is to find all words within a corpus that meet some criteria you are interested in. In this brief example, we will use character codes to search for words whose letters are "in order" when read from the first letter to the last. We will create a Boolean function `OrderedWordQ` that returns `True` or `False` depending upon whether its argument is in alphabetic order. So `OrderedWordQ["best"]` would return `True` but `OrderedWordQ["brag"]` would return `False`. Then we will use this predicate to find all words in a dictionary that are ordered in this sense.

Start by getting a list of all words in the dictionary using `DictionaryLookup`.

```
words = DictionaryLookup[];
Short[words, 4]
```

```
{a, Aachen, aah, Aaliyah, aardvark, aardvarks,
 Aaron, abaci, aback, abacus, abacuses, abaft,
 abalone, abalones, <<92491>>, Zunis, Zurich,
 Zürich, zwieback, Zwingli, Zworykin, zydeco,
 zygote, zygotes, zygotic, zymurgy, Zyrtec, Zyuganov}
```

Alternatively, you can use the data in `WordData`, which contains phrases in addition to words. You could use any similar resource for your list of words.

```
Short[WordData[All], 4]
```

```
{0, 1, 10, 100, 1000, 10000, 100000, 1000000,
 1000000000, 1000000000000, 1000th, 100th, 101, 101st,
 <<149163>>, zygote, zygotene, zygotic, Zyloprim,
 zymase, zymogen, zymoid, zymology, zymolysis,
 zymolytic, zymosis, zymotic, zymurgy, Zyrian}
```

First, consider the character code of a string.

```
ToCharacterCode["best"]
```

```
{98, 101, 115, 116}
```

Then we only need to know if this list of codes is in order.

```
OrderedQ[%]
```

```
True
```

Here is a predicate that returns `True` if its argument is ordered in this alphabetic sense.

```
OrderedWordQ[word_String] := OrderedQ[ToCharacterCode[word]]
```

Now we will find all the words in the dictionary file that comes with *Mathematica* that are ordered in this way; we will use `Select` to return those words that pass the test. Finally, we randomly sample 40 of them.

```
orderedwords = Select[words, OrderedWordQ];
```

```
RandomSample[orderedwords, 40]
```

```
{Cox, hit, chill, cw, Ibo, Uccello, dhow, Nair, Lent, Kenny,
 my, Gamow, BBC, alloy, lox, ls, Pacino, Sade, adios, lost,
 Chimu, bills, apt, or, any, doors, Topsy, Finn, May, Mai,
 know, dory, ills, boor, been, egg, floss, AD, bop, ells}
```

Almost correct! In the English character code set, capitals appear before lowercase letters. So, although our words are ordered in the sense of character codes, they are not ordered in the commonly-used sense.

```
ToCharacterCode["A"]
```

```
{65}
```

```
ToCharacterCode["a"]
```

```
{97}
```

One approach to resolving this issue is to only work with words of the same case. We could either convert words of the form uppercase/lowercase to lowercase/lowercase or we could select only words from the dictionary that match a pattern that codes for this. We will wait until the discussion of string patterns in Section 9.3 to correct this issue.

■ *Exercises*

1. Convert the first character in a string (which you may assume to be a lowercase letter) to uppercase.
2. Given a string of digits of arbitrary length, convert it to its integer value. (Hint: you may find that the `Dot` function is helpful.)
3. Create a function `UniqueCharacters[`*str*`]` that takes a string as its argument and returns a list of the unique characters in that string. For example, `UniqueCharacters["Mississippi"]` should return `{M, i, s, p}`.

■ 9.2. Operations on Strings

Strings are expressions and, like other expressions (such as numbers and lists), there are built-in functions available to operate on them. Many of these functions are very similar to those for operating on lists. In this section we will first look at some of these basic functions for operating on strings and then use them on some nontrivial examples: analyzing a large piece of text, encoding strings, creating index variables, and finally, a word game for creating anagrams.

□ Basic String Operations

`StringTake`, which has a similar syntax to `Take`, is used to extract parts of a string. The second argument specifies the positions of the characters to extract. So, for example, this takes the first twelve characters in this string.

```
StringTake["Three quarks for Muster Mark!", 12]
```

```
Three quarks
```

And this takes the last twelve characters from the string.

```
StringTake["Three quarks for Muster Mark!", -12]
```

```
Muster Mark!
```

A list of the individual characters is returned by `Characters`.

```
Characters["Three quarks for Muster Mark!"]
```

```
{T, h, r, e, e,  , q, u, a, r, k, s,  ,
 f, o, r,  , M, u, s, t, e, r,  , M, a, r, k, !}
```

`StringJoin` concatenates strings.

```
StringJoin["q", "u", "a", "r", "k", "s"]
```

```
quarks
```

The shorthand notation for `StringJoin` is str_1 < > str_2.

```
"x" <> "22"
```

```
x22
```

The following functions mirror those for list operations.

```
StringReverse["abcde"]

edcba

StringDrop["abcde", -1]

abcd

StringPosition["abcde", "bc"]

{{2, 3}}

StringCount["When you wish upon a star", "o"]

2

StringInsert["abcde", "T", 3]

abTcde

StringReplace["abcde", "cd" → "CD"]

abCDe
```

Some functions are quite specific to strings and do not have analogs with lists. For example, conversion to uppercase and lowercase.

```
ToUpperCase["words"]

WORDS
```

This trims substrings from a string using alternative patterns (discussed further in Section 9.3). So if either `"http://"` or `"/"` is found, they will be trimmed.

```
    StringTrim["http://www.google.com/", "http://" | "/"]

www.google.com
```

□ Strings vs. Lists

For some computations, you might be tempted to convert a string to a list of characters and then operate on the list using some list manipulation functions. For example, this first constructs a list of the individual characters and then uses `Count` to get the number of occurrences of the letter *B* in the list of characters from the text of Charles Darwin's *On the Origin of Species*.

```
text = ExampleData[{"Text", "OriginOfSpecies"}];
StringTake[text, 200]

INTRODUCTION. When on board H.M.S. 'Beagle,' as
  naturalist, I was much struck with certain facts
  in the distribution of the inhabitants of South
  America, and in the geological relations of the present
```

```
Count[Characters[text], "B"] // Timing

{0.242012, 427}
```

Since the string functions in *Mathematica* are optimized for working on strings directly you will often find that they are much faster than the more general list manipulation functions.

```
StringCount[text, "B"] // Timing

{0.001955, 427}
```

This speedup results from the fact that the string pattern matching algorithms are operating only on a well-defined finite alphabet and string expressions are essentially flat structures, whereas the algorithms for more general expression matching are designed to operate on arbitrary expressions with potentially much more complicated structures.

Converting to lists and using list manipulation functions will often be more cumbersome than working with the string functions directly. For example, counting the occurrences of a word within a chunk of text by first converting to a list of characters would be quite indirect and computationally more taxing than simply using `StringCount` directly.

```
StringCount[text, "selection"] // Timing

{0.006667, 351}
```

In fact, sometimes you will even find it more efficient to convert a numerical problem to one involving strings, do the work with string manipulation functions, and then convert back to numbers as in the subsequence example in Section 9.5.

□ Encoding Text

In this example, we will develop functions for coding and decoding strings of text. The particular coding that we will use is quite simplistic compared with contemporary commercial-grade ciphers, but it will give us a chance to see how to combine string manipulation, the use of functional programming constructs, and rule-based programming all in a very practical example that should be accessible to anyone.

The problem in encryption is to develop an algorithm that can be used to encode a string of text and then a dual algorithm that can be used to decode the encrypted message. Typically, the input string is referred to as the *plaintext* and the encoded output as the *ciphertext*.

To start, we will limit ourselves to the 26 lowercase letters of the alphabet.

```
alphabet = CharacterRange["a", "z"]
```

```
{a, b, c, d, e, f, g, h, i, j, k,
 l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

One of the simplest encryption schemes is attributed to Julius Caesar who is said to have used this cipher to encode communications with his generals. The scheme is simply to shift each letter of the alphabet some fixed number of places to the left and is commonly referred to as a substitution cipher. Using `Thread`, we can set up rules that implement this shift, here just shifting one place to the left.

```
CaesarCodeRules = Thread[alphabet → RotateLeft[alphabet]]
```

```
{a → b, b → c, c → d, d → e, e → f, f → g, g → h, h → i,
 i → j, j → k, k → l, l → m, m → n, n → o, o → p, p → q, q → r,
 r → s, s → t, t → u, u → v, v → w, w → x, x → y, y → z, z → a}
```

The decoding rules are simply to reverse the encoding rules.

```
CaesarDecodeRules = Map[Reverse, CaesarCodeRules]
```

```
{b → a, c → b, d → c, e → d, f → e, g → f, h → g, i → h,
 j → i, k → j, l → k, m → l, n → m, o → n, p → o, q → p, r → q,
 s → r, t → s, u → t, v → u, w → v, x → w, y → x, z → y, a → z}
```

To code a string, we will decompose the string into individual characters, apply the code rules, and then join up the resulting characters in a “word.”

```
Characters["hello"]
```

```
{h, e, l, l, o}
```

```
% /. CaesarCodeRules
```

```
{i, f, m, m, p}
```

```
StringJoin[%]
```

```
ifmmp
```

Here is the function to accomplish this.

```
encode[str_String, coderules_] :=
 StringJoin[Characters[str] /. coderules]
```

Similarly, here is the decoding function.

```
decode[str_String, decoderules_] :=
 StringJoin[Characters[str] /. decoderules]
```

Let us try it out on a phrase.

```
encode["squeamish ossifrage", CaesarCodeRules]
```

```
trvfbnjti pttjgsbhf
```

```
decode[%, CaesarDecodeRules]
```

```
squeamish ossifrage
```

In this example, we have shifted one position for each letter to encode (and decode). It is thought that Caesar (or his cryptographers) used a shift of length three to encode his military messages. In the exercises, you are asked to implement a different shift length in the encoding and decoding functions.

Even with longer shifts, the Caesar cipher is terribly insecure and highly prone to cracking since there are only 26 possible shifts with this simple cipher. A slightly more secure cipher involves permuting the letters of the alphabet.

```
p = RandomSample[alphabet]
```

```
{n, d, l, v, a, b, c, o, x, z, w,
 j, p, u, s, r, y, f, e, k, g, h, q, i, t, m}
```

Using `Thread`, we create a rule for each letter paired up with the corresponding letter from the permutation `p`.

```
PermutationCodeRules = Thread[alphabet → p]
```

```
{a → n, b → d, c → l, d → v, e → a, f → b, g → c, h → o,
 i → x, j → z, k → w, l → j, m → p, n → u, o → s, p → r, q → y,
 r → f, s → e, t → k, u → g, v → h, w → q, x → i, y → t, z → m}
```

Again, the decoding rules are obtained by simply reversing the above rules.

```
PermutationDecodeRules = Thread[p → alphabet]
```

```
{n → a, d → b, l → c, v → d, a → e, b → f, c → g, o → h,
 x → i, z → j, w → k, j → l, p → m, u → n, s → o, r → p, y → q,
 f → r, e → s, k → t, g → u, h → v, q → w, i → x, t → y, m → z}
```

```
encode["squeamish ossifrage", PermutationCodeRules]
```

```
eyganpxeo seexbfnca
```

```
decode[%, PermutationDecodeRules]
```

```
squeamish ossifrage
```

Although these substitution ciphers are not terribly difficult to crack, they should give you some good practice in working with strings and the various *Mathematica* programming constructs. Modern commercial-grade ciphers such as public-key ciphers are often based on the difficulty of factoring large integers. For a basic introduction to the history of ciphers, see Sinkov (1966) [2]. A more thorough treatment can be found in Paar and Pelzl (2010) [3].

□ Indexed Symbols

When developing algorithms that operate on large structures (for example, large systems of equations), it is often helpful to be able to create a set of unique symbols with which to work. As an example of operations on strings, we will use some of the functions discussed in this section to develop a little utility function that creates unique symbols. Although there is a built-in function, `Unique`, that does this, it has some limitations for this particular task.

```
Table[Unique["x"], {8}]
```

```
{x5, x6, x7, x8, x9, x10, x12, x13}
```

One potential limitation of `Unique` is that it uses the first *unused* symbol of a particular form. It does this to avoid overwriting existing symbols.

```
Table[Unique["x"], {8}]

{x14, x15, x16, x17, x18, x19, x20, x21}
```

However, if you want to explicitly create a list of indexed symbols with a set of specific indices, it is useful to create a different function. First, note that a string can be converted to a symbol using `ToExpression` or by wrapping the string in `Symbol`.

```
Head["x1"]

String

ToExpression["x1"] // Head

Symbol

Symbol["x1"] // Head

Symbol
```

`StringJoin` is used to concatenate strings. So, let us concatenate the variable with the index, first with one number and then with a range of numbers.

```
StringJoin["x", "8"] // FullForm

"x8"

ToExpression[Map["x" <> ToString[#] &, Range[12]]]

{x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12}
```

We put all the pieces of code together.

```
MakeVarList[x_Symbol, n_Integer] :=
 ToExpression[Map[ToString[x] <> ToString[#] &, Range[n]]]

MakeVarList[tmp, 20]

{tmp1, tmp2, tmp3, tmp4, tmp5, tmp6,
 tmp7, tmp8, tmp9, tmp10, tmp11, tmp12, tmp13,
 tmp14, tmp15, tmp16, tmp17, tmp18, tmp19, tmp20}
```

Let us create an additional rule for this function that takes a range specification as its second argument.

```
MakeVarList[x_Symbol, {n_Integer, m_Integer}] :=
 ToExpression[Map[ToString[x] <> ToString[#] &, Range[n, m]]]
```

```
MakeVarList[tmp, {20, 30}]
```

```
{tmp20, tmp21, tmp22, tmp23, tmp24,
 tmp25, tmp26, tmp27, tmp28, tmp29, tmp30}
```

Note that we have not been too careful about argument checking.

```
MakeVarList[tmp, {-2, 2}]
```

```
{-2 + tmp, -1 + tmp, tmp0, tmp1, tmp2}
```

In the exercises you are asked to correct this.

□ Anagrams

Anagrams are words that have the same set of letters but in a different order. Good Scrabble players are adept at anagram creation. Anagrams can be created by taking a word, extracting and permuting its characters, and then finding which permutations are real words.

Start by getting the characters in a word.

```
chars = Characters["tame"]
```

```
{t, a, m, e}
```

Permute the characters.

```
p = Permutations[chars]
```

```
{{t, a, m, e}, {t, a, e, m}, {t, m, a, e}, {t, m, e, a},
 {t, e, a, m}, {t, e, m, a}, {a, t, m, e}, {a, t, e, m},
 {a, m, t, e}, {a, m, e, t}, {a, e, t, m}, {a, e, m, t},
 {m, t, a, e}, {m, t, e, a}, {m, a, t, e}, {m, a, e, t},
 {m, e, t, a}, {m, e, a, t}, {e, t, a, m}, {e, t, m, a},
 {e, a, t, m}, {e, a, m, t}, {e, m, t, a}, {e, m, a, t}}
```

Concatenate the characters in each list.

```
words = Map[StringJoin, p]

{tame, taem, tmae, tmea, team, tema, atme, atem,
 amte, amet, aetm, aemt, mtae, mtea, mate, maet,
 meta, meat, etam, etma, eatm, eamt, emta, emat}
```

Now, which of these "words" are really words? One way to check is to select those that are in the dictionary. Those elements in `words` that are not in the dictionary will return { } when run against `DictionaryLookup`, so we omit those using ≠.

```
Select[words, DictionaryLookup[#, IgnoreCase → True] ≠ {} &]

{tame, team, mate, meta, meat}
```

Putting all the pieces together, we have the function `Anagrams`.

```
Anagrams[word_String] :=
 Module[{chars = Characters[word], words},
  words = Map[StringJoin, Permutations[chars]];
  Select[words,
   DictionaryLookup[#, IgnoreCase → True] ≠ {} &]
 ]

Anagrams["parsley"] // Timing

{0.429069, {parsley, parleys, players, replays, sparely}}

Anagrams["elvis"]

{elvis, evils, levis, lives, veils}

Anagrams["instance"]

{instance, ancients, canniest}
```

Other than extracting the characters of a word and joining the permuted list of characters, the operations here are essentially those on lists (of strings) and pattern matching. Exercise 2 in Section 9.5 discusses a more direct approach to this problem, one that avoids the creation of permutations of the characters in the word.

■ *Exercises*

1. Create a function `PalindromeQ[`*str*`]` that returns a value of `True` if its argument *str* is a palindrome, that is, if the string *str* is the same forward and backward. For example, *refer* is a palindrome.

2. Create a function `stringRotateLeft[`*str*`, `*n*`]` that takes a string *str*, and returns a string with the characters rotated to the left *n* places. For example:

```
stringRotateLeft["a quark for Muster Mark ", 8]
```

```
for Muster Mark a quark
```

3. In creating the function `MakeVarList` in this section, we were not careful about the arguments that might be passed. Correct this problem using pattern matching on the arguments to this function to insure that the indices are positive integers only.

4. Create a function `StringPad[`*str*`, {`*n*`}]` that pads the end of a string with *n* whitespace characters. Then create a second rule `StringPad[`*str*`, `*n*`]` that pads the string out to length *n*. If the input string has length greater than *n*, issue a warning message. Finally, mirroring the argument structure for the built-in `PadLeft`, create a third rule `StringPad[`*str*`, `*n*`, `*m*`]` that pads with *n* whitespaces at the front and *m* whitespaces at the end of the string.

5. Modify the Caesar cipher so that it encodes by shifting five places to the right. Include the space character in the alphabet.

6. A mixed-alphabet cipher is created by first writing a keyword followed by the remaining letters of the alphabet and then using this as the substitution (or cipher) text. For example, if the keyword is *django*, the cipher text alphabet would be:

   `djangobcefhiklmpqrstuvwxyz`

   So, *a* is replaced with *d*, *b* is replaced with *j*, *c* is replaced with *a*, and so on. As an example, the piece of text

   the sheik of araby

   would then be encoded as

   tcg scgeh mo drdjy

   Implement this cipher and go one step further to output the cipher text in blocks of length five, omitting spaces and punctuation.

7. Modify the alphabet permutation cipher so that instead of being based on single letters, it is instead based on adjacent pairs of letters. The single letter cipher will have $26! = 403\,291\,461\,126\,605\,635\,584\,000\,000$ permutations; the adjacent pairs cipher will have $26^2! = 1.883707684133810\ 10^{1621}$ permutations.

■ 9.3. String Patterns

Most of the string operations we have looked at up until this point have involved literal strings. For example, in string replacement, we have specified both the explicit string that we are operating on as well as the replacement string.

```
StringReplace["11/28/1986", "/" → "-"]
```

```
11-28-1986
```

But the real power of programming with strings comes with the use of patterns to represent different classes of strings. A string pattern is a string expression that contains symbolic patterns. Much of the pattern matching discussed in the previous chapters extends to strings in a very powerful manner. For example, this uses patterns to change the first letter in a string to uppercase.

```
str = "colorless green ideas sleep furiously";
```

```
StringReplace[str, f_ ~~ rest__ :→ ToUpperCase[f] <> rest]
```

```
Colorless green ideas sleep furiously
```

Or, use a conditional pattern to check if a word begins with an uppercase character.

```
StringMatchQ["Jekyll", f_?UpperCaseQ ~~ rest___]
```

```
True
```

To get started, you might find it helpful to think of strings as a sequence of characters and use the same general principles on these expressions as you do with lists.

For example, the expression `{a, b, c, c, d, e}` matches the pattern `{__, s_, s_, __}` because it is a list that starts with a sequence of one or more elements, it contains an element repeated once, and then ends with a sequence of one or more elements.

```
MatchQ[{a, b, c, c, d, e}, {__, s_, s_, __}]
```

```
True
```

If we now use a string instead of a list and `StringMatchQ` instead of `MatchQ`, we get a similar result using the shorthand notation ~~ for `StringExpression`.

```
StringMatchQ["abccde", __ ~~ s_ ~~ s_ ~~ __]
```

```
True
```

str_1 ~~ str_2 is shorthand notation for `StringExpression[`str_1 ~~ str_2`]`, which, for the purpose of pattern matching, represents a sequence of strings.

```
"a" ~~ "b"
```

```
ab
```

```
Defer[FullForm["a" ~~ "b"]]
```

```
StringExpression["a", "b"]
```

`StringExpression` is quite similar to `StringJoin` (both can be used to concatenate strings) except that with `StringExpression`, you can concatenate nonstrings.

The next example also shows the similarity between the general expression pattern matching that we explored earlier in Chapter 4 and string patterns. Using `Cases`, the following returns all those expressions that match the pattern `_Symbol`, that is, pick out all symbols from the list.

```
Cases[{1, f, g, 6, x, t, 2, 5}, _Symbol]
```

```
{f, g, x, t}
```

With strings we use `StringCases` whose second argument is a pattern that represents a class of characters to match. `StringCases` returns those substrings that match a given pattern. Many named patterns are available for various purposes. For example, `LetterCharacter` matches a single letter.

```
StringCases["1fg6xt25", LetterCharacter]
```

```
{f, g, x, t}
```

Match single digits with `DigitCharacter` and one or more digits with `NumberString`.

```
StringCases["1fg6xt25", DigitCharacter]
```

```
{1, 6, 2, 5}
```

```
StringCases["1fg6xt25", NumberString]
```

```
{1, 6, 25}
```

To see the generality and power of working with string patterns, suppose we were looking for a nucleotide sequence in a gene consisting of a repetition of A followed by any character, followed by T. Using a gene from the human genome, the following string pattern neatly does the job.

```
gene = GenomeData["IGHV357"]
```

```
AAGTCCTGTGTGAAGTTTATTGATGGAGTCAGAGGCAGAAAATTGTACAGCCCAGTGGT⋱
    TCACTGAGACTCTCCTGCAAAGCCTCTGATTTCACCTTTACTGGCTACAGCATGA⋱
    GCTTGGTCCAGCAGGCTTCATGACAGGGATTGGTGTGGGTGGAAACAGTGAGTGA⋱
    TCAAGTGGGAGTTCTCAGAGTTACTCTCCATGAGTACAAATAAATTAACAGTCCC⋱
    AAGCGACACCTTTTCATGTGCAGTCTACCTTACAATGACCAACCTGAAAGCCAAG⋱
    GACAAGGCTGTGTATTACTGTGAGGGA
```

```
StringCases[gene, "AA" ~~ _ ~~ "T"]
```

```
{AAGT, AAGT, AAAT, AAGT, AAAT, AAAT}
```

Here are the starting and ending positions of these substrings. `StringPosition` takes the same syntax as `StringCases`, analogous to `Position` and `Cases`.

```
StringPosition[gene, "AA" ~~ _ ~~ "T"]
```

```
{{1, 4}, {13, 16}, {40, 43}, {41, 44},
  {172, 175}, {207, 210}, {211, 214}, {212, 215}}
```

And if you wanted to return those characters that follow all occurrences of the string `"GTC"`, you can name the pattern and use a rule to return it.

```
StringCases[gene, pat : "GTC" ~~ x_ :→ pat <> x]
```

```
{GTCC, GTCA, GTCC, GTCC, GTCT}
```

In this example, the pattern is `pat : "GTC" ~~ x_`. This pattern is named `pat` and it consists of the string GTC which is then followed by any character. That character is named `x` so that we can refer to it in the replacement expression on the right-hand side of the rule. The replacement expression is the pattern `pat` concatenated with the character named `x`.

As another example of the use of string patterns, suppose you were interested in scraping phone numbers off of a web page; you need to construct a pattern that matches the form of the phone numbers you are looking for. In this case we use the form $n - nnn - nnn - nnnn$ which matches the form of North American phone numbers. `NumberString` comes in handy as it picks up strings of numbers of any length. Otherwise you would have to use `DigitCharacter ..` which matches repeating digits.

```
webpage = Import["http://www.wolfram.com/company/contact.cgi",
   "HTML"];
```

```
StringCases[webpage,
 NumberString ~~ "-" ~~ NumberString ~~ "-" ~~ NumberString ~~
  "-" ~~ NumberString ]
```

```
{+1-217-398-0700, +1-217-398-0747}
```

□ Finding Subsequences with Strings

In this section we will explore a related problem to the one in Section 4.3, where we searched for subsequences within a sequence of numbers. Here we will transform the problem from working with lists of digits to one where we work with strings.

Using pattern matching it is not too difficult to construct the pattern of interest. For example, suppose we were looking for the substring *are* within a larger string. Using the special named string pattern `WordBoundary` which matches the beginning or end of a word, we concatenate (`StringJoin`) the patterns we need. See Table 9.3 in the next section for a listing of other named patterns.

```
StringCases["The magic words are squeamish ossifrage.",
 WordBoundary ~~ "are" ~~ WordBoundary]
```

```
{are}
```

```
StringPosition["The magic words are squeamish ossifrage.",
 WordBoundary ~~ "are" ~~ WordBoundary]
```

```
{{17, 19}}
```

To start, we will prototype with a short sequence of digits of π, converted to a string.

```
num = ToString[N[π, 50]]
```

```
3.1415926535897932384626433832795028841971693993751
```

Check that the output is in fact a string.

```
{Head[num], InputForm[num]}
```

```
{String,
 "3.1415926535897932384626433832795028841971693993751"}
```

For our purposes here, we are only interested in the digits following the decimal point. We can extract them by splitting the string of digits on the decimal point and then taking the second part of that expression. This will generalize for numbers with an arbitrary number of digits before the decimal point.

```
StringSplit[num, "."]
```

```
{3, 1415926535897932384626433832795028841971693993751}
```

```
Part[%, 2]
```

```
1415926535897932384626433832795028841971693993751
```

The subsequence 3238 occurs starting 15 positions to the right of the decimal point.

```
StringPosition[%, "3238"]
```

```
{{15, 18}}
```

Collecting the code fragments, we turn this into a function.

```
FindSubsequence[num_?NumberQ, subseq_?NumberQ] :=
 With[{n = ToString[num], s = ToString[subseq]},
  StringPosition[Part[StringSplit[n, "."], 2], s]
 ]
```

Let us try it out on a more challenging example: finding occurrences of the sequence 314159 in the decimal expansion of π to ten million places.

```
pi = N[π, 10^7];
```

```
FindSubsequence[pi, 314 159] // Timing
```

```
{5.688071, {{176 451, 176 456},
  {1 259 351, 1 259 356}, {1 761 051, 1 761 056},
  {6 467 324, 6 467 329}, {6 518 294, 6 518 299},
  {9 753 731, 9 753 736}, {9 973 760, 9 973 765}}}
```

Comparing with the function that takes lists of digits developed in Section 4.3, our string implementation is about twice as fast.

```
FindSubsequence2[digits_List, subseq_List] :=
 Module[{p, len = Length[subseq]},
  p = Partition[digits, len, 1];
  Position[p, subseq] /. {num_?IntegerQ} :→ {num, num + len - 1}]

pidigs = First[RealDigits[π, 10, 10^7, -1]];
Timing[FindSubsequence2[pidigs, {3, 1, 4, 1, 5, 9}]]

{9.494993, {{176 451, 176 456},
  {1 259 351, 1 259 356}, {1 761 051, 1 761 056},
  {6 467 324, 6 467 329}, {6 518 294, 6 518 299},
  {9 753 731, 9 753 736}, {9 973 760, 9 973 765}}}
```

□ Alternatives

We have already seen general patterns with alternatives discussed in Chapter 4. Here we will use alternatives with string patterns. The idea is quite similar. For example, a common task in genome analysis is determining the GC content or ratios of the nucleobases guanine (G) and cytosine (C) to all four bases in a given fragment of genetic material.

```
gene = GenomeData["MRPS35P1"];
```

You could count the occurrences of G and the occurrences of C and add them together.

```
StringCount[gene, "G"] + StringCount[gene, "C"]

41
```

But it is much easier to use alternatives to indicate that you want to count all occurrences of either G or C. The syntax for using alternative string patterns is identical to that for general expressions that we introduced in Section 4.1.

```
StringCount[gene, "G" | "C"]

41
```

We will return to the computation of GC content in Section 9.5.

As a slightly more involved example, suppose you are interested in tallying the lengths of words in a corpus. You might start by using `StringSplit` to split the large string into a list of words.

```
text = ExampleData[{"Text", "OriginOfSpecies"}];
```

```
sstext = StringSplit[text];
Short[sstext, 6]
```

```
{INTRODUCTION., When, on, board, H.M.S., 'Beagle,', as,
 naturalist,, I, was, much, struck, with, certain, facts, in,
 the, distribution, of, the, inhabitants, of, South, America,,
 <<149 815>>, to, the, fixed, law, of, gravity,, from, so,
 simple, a, beginning, endless, forms, most, beautiful, and,
 most, wonderful, have, been,, and, are, being,, evolved.}
```

Looking at the result, you will see that some elements of this list include various types of punctuation. For example, `StringSplit`, with default delimiters, missed certain hyphenated words and some punctuation.

```
sstext[[{53, 362}]]
```

```
{species--that, statements;}
```

There are 149863 elements in this split list.

```
Length[sstext]
```

```
149 863
```

Fortunately, `StringSplit` takes a second argument that specifies the delimiters to match. The pattern is given as a set of alternatives followed by the repeated operator to catch one or more repetitions of any of these delimiters. Searching through the text will help to come up with this list of alternatives.

```
splitText = StringSplit[text,
   (" " | "." | "," | ";" | ":" | "'" | "\"" | "?" | "!" | "-") ..];
```

```
Short[splitText, 5]
```

```
{INTRODUCTION, When, on, board, H, M, S, Beagle, as,
 naturalist, I, was, much, struck, with, certain, facts, in,
 the, distribution, of, <<151 160>>, law, of, gravity, from,
 so, simple, a, beginning, endless, forms, most, beautiful,
 and, most, wonderful, have, been, and, are, being, evolved}
```

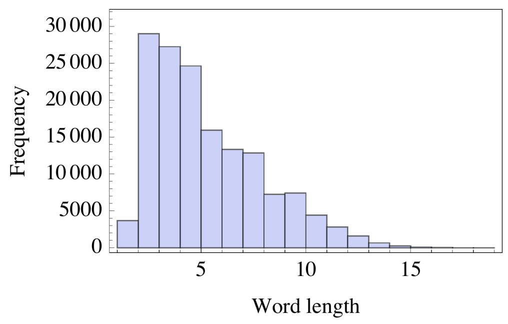
Notice that this list contains many more elements than the initial approach given above.

```
Length[splitText]
```

```
151 202
```

Finally, here is a histogram showing the distribution of word lengths in the text, *On the Origin of Species*.

```
Histogram[StringLength[splitText], Frame → True,
 FrameLabel → {"Word length", "Frequency"},
 FrameTicks → {{Automatic, None}, {Automatic, None}}]
```



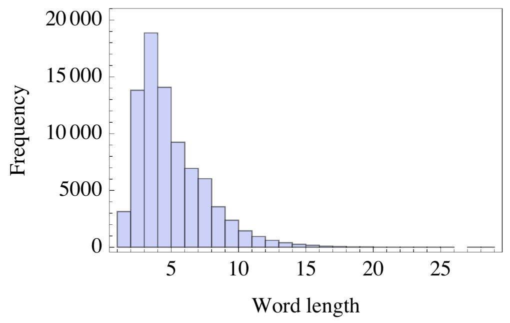
Let us compare this with a different text: *A Portrait of the Artist as a Young Man*, by James Joyce [4] (available online at Project Gutenberg [5]). We are postprocessing here by removing metadata at the beginning and at the end of the file.

```
joyce =
  StringTake[
   Import[
    "http://www.gutenberg.org/cache/epub/4217/pg4217.txt",
    "Text"], 688 ;; -18 843];
StringTake[joyce, {75, 164}]
```

```
Once upon a time and a very
  good time it was there was a moocow coming
down along the road
```

An alternative syntax uses a list of delimiters as given by `Characters`. The repeated pattern, `..`, helps to catch such constructions as "`--`", "`::`" and double-spaces.

```
words = StringSplit[joyce, Characters[":,;.!?'\- "] ..];
Histogram[StringLength[words], Frame → True,
 FrameLabel → {"Word length", "Frequency"},
 FrameTicks → {{Automatic, None}, {Automatic, None}}]
```



In the next section, on regular expressions, we will see that there are more compact ways of accomplishing some of these tasks.

▪ *Exercises*

1. At the end of Section 9.1 we created a predicate `OrderedWordQ` to find all words in a dictionary whose letters are in alphabetic order. This predicate used character codes and returned incorrect results for words that started with a capital letter. Correct this error by only selecting words from the dictionary that start with a lowercase letter. Consider using a conditional string pattern involving the built-in function `LowerCaseQ`.
2. Given a list of words, some of which start with uppercase characters, convert them all to words in which the first character is lowercase. You can use the words in the dictionary as a good sample set.
3. Create a function `Palindromes[`*n*`]` that finds all palindromic words of length *n* in the dictionary. For example, *kayak* is a five-letter palindrome.
4. Find the number of unique words in a body of text such as *Alice in Wonderland*.

   ```
   text = ExampleData[{"Text", "AliceInWonderland"}];
   ```

   After splitting the text into words, convert all uppercase characters to lowercase so that you count words such as *hare* and *Hare* as the same word.

   Such computations are important in information retrieval systems, for example, in building term-document incidence matrices used to compare the occurrence of certain terms across a set of documents (Manning, Raghavan, and Schütze 2008 [6]).

■ 9.4. Regular Expressions

In addition to the use of string patterns discussed up to this point, you can also specify string patterns using what are known as *regular expressions*. Regular expressions in *Mathematica* follow a syntax very close to that of the Perl programming language. This syntax is quite compact and powerful but it comes at the cost of readability – regular expressions tend to be quite cryptic to humans. As a result, we will only cover a few examples of their use here and refer the interested reader to the *Mathematica* documentation on string patterns (Working with String Patterns, WMDC [7]).

You should think of regular expressions as an alternative syntax for string pattens. To indicate that you are using a regular expression, wrap the expression in `RegularExpression`. For example, the regular expression **.** is a wildcard character. It matches any single character except a newline. To use it as a string pattern, write `RegularExpression["."]`.

```
StringMatchQ["a", RegularExpression["."]]

True
```

The string `"abc"` does not match the pattern because it does not consist of a single character.

```
StringMatchQ["abc", RegularExpression["."]]

False
```

You can also match a set or range of characters. For example, this matches any of the characters *a* through *z*.

```
StringMatchQ["a", RegularExpression["[a-z]"]]

True
```

Certain constructs give patterns with repeating elements. For example, `"c*"` is a pattern matched by a string with character *c* repeated zero or more times; `"c+"` stands in for the character *c* repeated one or more times.

```
StringMatchQ["aa", RegularExpression["a*"]]

True
```

```
StringMatchQ["aaab", RegularExpression["a+"]]

False
```

You can also match on concatenated characters using the syntax c_1 c_2 ….

```
StringPosition["ACAACTGGAGATCATGACTG",
 RegularExpression["ACT"]]
```

```
{{4, 6}, {17, 19}}
```

Several constructs are available for classes of characters. The named classes in the last two entries of Table 9.2 include *alpha*, *ascii*, *blank*, *digit*, *space*, *word*, and several more.

Regular expression	Meaning
`\\d`	digit 0 – 9
`\\D`	nondigit
`\\s`	space, newline, tab, whitespace
`\\S`	non-whitespace character
`\\w`	word character, e.g. letter, digit
`\\W`	nonword character
`[[:`*class*`:]]`	characters in a named class
`[^[:`*class*`:]]`	characters not in a named class

▲ **Table 9.2.** Regular expressions classes of characters

The regular expression `a.*` matches any expression beginning with the character *a* followed by any sequence of characters.

```
StringMatchQ["all in good time", RegularExpression["a.*"]]
```

```
True
```

The regular expression `\\ d` represents any digit 0 through 9.

```
StringCases["1a2b3c4d", RegularExpression["\\d"]]
```

```
{1, 2, 3, 4}
```

The regular expression `a.+\\ d` matches any expression beginning with an *a*, followed by any character repeated one or more times, followed by a digit.

```
StringCases["abc1, abd2, abc", RegularExpression["a.+\\d"]]
```

```
{abc1, abd2}
```

Let us try something more ambitious. This finds all words in `text` that are of length 16 to 18.

```
text = ExampleData[{"Text", "OriginOfSpecies"}];
StringCases[text,
  RegularExpression["\\b\\w{16,18}\\b"]] // DeleteDuplicates
```

```
{agriculturalists, disproportionably,
 malconformations, experimentalists, palaeontological,
 incomprehensibly, PALAEONTOLOGICAL, palaeontologists,
 intercommunication, incomprehensible}
```

The regular expression `\\ b` matches any word boundary (typically whitespace, period, comma, etc.) and `\\ w {16, 18}` matches any word of length 16 to 18.

Various shortcuts exist for some commonly used patterns (Table 9.3).

Pattern	Matches
`StartOfString`	beginning of entire string
`EndOfString`	end of entire string
`StartOfLine`	beginning of a line
`EndOfLine`	end of a line
`WordBoundary`	boundary between words

▲ **Table 9.3.** Patterns for special locations within strings

Conveniently, you can mix regular expressions and other string patterns in various ways. This accomplishes the same thing as the previous computation, but using `WordBoundary`, instead of the regular expression `\\ b`.

```
StringCases[text,
  WordBoundary ~~ RegularExpression["\\w{16,18}"] ~~
   WordBoundary] // DeleteDuplicates
```

```
{agriculturalists, disproportionably,
 malconformations, experimentalists, palaeontological,
 incomprehensibly, PALAEONTOLOGICAL, palaeontologists,
 intercommunication, incomprehensible}
```

Sometimes you will need to refer to the pattern by name in order to perform some operation on it. This is similar to the situation with regular named patterns. For example, given a list of words, some of which are uppercase/lowercase, this uses string patterns to transform the list to all lowercase words, naming the pattern that is matched by the first character after a word boundary, `a`.

```
words = {"festively", "frolicking", "subcategories",
   "retreated", "recompiling", "Barbary", "Herefords",
   "geldings", "Norbert", "incalculably", "proselytizers",
   "topmast"};
```

```
StringReplace[words, WordBoundary ~~ a_ :> ToLowerCase[a]]
```

```
{festively, frolicking, subcategories, retreated,
 recompiling, barbary, herefords, geldings,
 norbert, incalculably, proselytizers, topmast}
```

So how do we name a pattern with regular expressions so that we can refer to it on the right-hand side of a rule? The syntax using regular expressions is to wrap the pattern in parentheses and then refer to it using `"$`*n*`"`, where *n* is the *n*th occurrence of such patterns. For example, `\\ b (\\ w)` is a named pattern that is matched by an expression consisting of a word boundary followed by a word character. The subexpression matching `(\\ w)` is referenced by `"$1"` on the right-hand side of the rule.

```
StringReplace[words,
 RegularExpression["\\b(\\w)"] :> ToLowerCase["$1"]]
```

```
{festively, frolicking, subcategories, retreated,
 recompiling, barbary, herefords, geldings,
 norbert, incalculably, proselytizers, topmast}
```

To change the second character after the word boundary to uppercase, use `"$2"` to refer to the expression that matches the second `(\\ w)`.

```
StringReplace[words,
 RegularExpression["\\b(\\w)(\\w)"] :>
  ToLowerCase["$1"] ~~ ToUpperCase["$2"]]
```

```
{fEstively, fRolicking, sUbcategories, rEtreated,
 rEcompiling, bArbary, hErefords, gEldings,
 nOrbert, iNcalculably, pRoselytizers, tOpmast}
```

A particularly useful construct in many situations is the lookahead/lookbehind construct. (**?** = *patt*) is used when the following text must match *patt* and (**?** < = *patt*) is used when the preceding text must match *patt*. For example, this finds all those words in some example text that follow `"Raven, "`.

```
text = ExampleData[{"Text", "TheRaven"}];

StringCases[text,
 RegularExpression["(?<=Raven, )\\w+"]]

{sitting, never}
```

There are many more constructs available for doing quite sophisticated things with regular expressions. We will explore some of these in the examples and exercises below as well as in the applications in Section 9.5. For a more detailed discussion, see the tutorials Regular Expressions (WMDC [8]) and Working with String Patterns (WMDC [7]).

□ Word Stemming

Many aspects of linguistic analysis include a study of the words used in a piece of text or in a speech. For example, you might be interested in comparing the complexity of articles written in two different newspapers. The length and frequency of certain words might be a useful measure for such an analysis. Patterns in usage of certain word combinations can be used to identify authenticity or the identity of an author of a work.

There are some basic issues that arise again and again in such analyses. For example, what should be done with contractions such as *shouldn't*? What about sets of words such as *run*, *runs*, *ran*, *running*. Are they considered distinct? One approach in language processing is to strip suffixes and reduce alternate forms to some *stem*. This process, known as word stemming, is extensively used in many online search systems to try to distill user's queries to some basic form that can be processed and operated on. It is a bit tricky, as natural languages are notorious for exceptions to almost any rule. For example, although the word *entertainment* can sensibly be stemmed to *entertain*, the stem of *comment* is certainly not *com*. In other words, a rule that dropped the suffix *ment* is too broad and returns nonwords in many cases. In most word stemming algorithms, there are numerous rules for the many cases that need to be examined; and there are many special cases. In this section, we will create a set of rules for word stemming to show how these rules are described and how the string pattern constructs in *Mathematica* provide a good set of tools to implement these concepts. A full-fledged stemming application would include hundreds of rules for each language, so we will only give a small set here to indicate the general process.

Words Ending in ...xes

The first set of stemming rules we will create involves a relatively small set of words in the English language – those ending in *xes*, such as *boxes* or *complexes*. The rule is to strip off the *es*.

To prototype, we collect all the words in the dictionary that end in *xes*. We will also restrict ourselves to words that are all lowercase. `Quiet` is used here to suppress the error messages that arise when `StringTake` operates on words of length less than three. Alternatively, you could put an extra clause (`StringLength[w] ≥ 3`) in the conjunction.

```
words =
 DictionaryLookup[
   w__ /; StringTake[w, -3] === "xes" &&
     LowerCaseQ@StringTake[w, 1]] // Quiet
```

```
{admixes, affixes, annexes, anticlimaxes, apexes, appendixes,
 aviatrixes, axes, bandboxes, bollixes, boxes, breadboxes,
 calyxes, chatterboxes, circumflexes, climaxes, coaxes,
 complexes, convexes, coxes, crucifixes, cruxes, detoxes,
 duplexes, equinoxes, exes, faxes, fireboxes, fixes,
 flexes, flummoxes, fluxes, foxes, gearboxes, hatboxes,
 hexes, hoaxes, horseboxes, hotboxes, ibexes, iceboxes,
 indexes, influxes, intermixes, jinxes, jukeboxes, laxes,
 letterboxes, loxes, lummoxes, lunchboxes, lynxes,
 mailboxes, matchboxes, maxes, minxes, mixes, moneyboxes,
 multiplexes, nixes, onyxes, orthodoxes, outboxes, outfoxes,
 overtaxes, oxes, paintboxes, paradoxes, parallaxes,
 perplexes, phalanxes, phoenixes, pickaxes, pillboxes,
 pixes, poleaxes, postboxes, postfixes, poxes, prefixes,
 premixes, prophylaxes, pyxes, reflexes, relaxes, remixes,
 saltboxes, sandboxes, saxes, sexes, shadowboxes, simplexes,
 sixes, snuffboxes, soapboxes, sphinxes, squeezeboxes,
 strongboxes, suffixes, surtaxes, taxes, telexes, thoraxes,
 tinderboxes, tippexes, toolboxes, transfixes, triplexes,
 tuxes, unfixes, vertexes, vexes, vortexes, waxes, xeroxes}
```

Here is the replacement rule. The regular expression `"(\\w)(x)es"` will be matched by any word character followed by *xes*. It is replaced by that word character followed only by *x*. On the right-hand side of the rule, `$1` refers to the first pattern on the left, `(\\w)`; and `$2` refers to the second pattern on the left, `(x)`.

```
rule1 = RegularExpression["(\\w)(x)es"] :→ "$1$2";
```

```
stemmed = StringReplace[words, rule1]

{admix, affix, annex, anticlimax, apex, appendix, aviatrix,
 ax, bandbox, bollix, box, breadbox, calyx, chatterbox,
 circumflex, climax, coax, complex, convex, cox, crucifix,
 crux, detox, duplex, equinox, ex, fax, firebox, fix, flex,
 flummox, flux, fox, gearbox, hatbox, hex, hoax, horsebox,
 hotbox, ibex, icebox, index, influx, intermix, jinx,
 jukebox, lax, letterbox, lox, lummox, lunchbox, lynx,
 mailbox, matchbox, max, minx, mix, moneybox, multiplex,
 nix, onyx, orthodox, outbox, outfox, overtax, ox, paintbox,
 paradox, parallax, perplex, phalanx, phoenix, pickax,
 pillbox, pix, poleax, postbox, postfix, pox, prefix,
 premix, prophylax, pyx, reflex, relax, remix, saltbox,
 sandbox, sax, sex, shadowbox, simplex, six, snuffbox,
 soapbox, sphinx, squeezebox, strongbox, suffix, surtax,
 tax, telex, thorax, tinderbox, tippex, toolbox, transfix,
 triplex, tux, unfix, vertex, vex, vortex, wax, xerox}

Select[stemmed, Not@MemberQ[DictionaryLookup[], #] &]

{max, poleax, postfix, prophylax}
```

This is pretty good; it appears only four stemmed words are not in the dictionary; although *max* might be considered an abbreviation, *postfix* is certainly a word! Nonetheless, these sorts of exceptions are common and will need to be dealt with separately.

■ *Plural Nouns Ending …mming*

A word such as *programming* has a stem of *program*; so the rule for words ending in …*mming* could be: drop the *ming*. Start by gathering all the words in the dictionary that end with …*mming*.

```
words =
 DictionaryLookup[w__ /; StringTake[w, -5] === "mming"] //
  Quiet

{bedimming, brimming, bumming, chumming, clamming, cramming,
 damming, deprogramming, diagramming, dimming, drumming,
 flimflamming, gumming, hamming, hemming, humming, jamming,
 lamming, lemming, monogramming, multiprogramming,
 programming, ramming, reprogramming, rimming, scamming,
 scramming, scrumming, scumming, shamming, shimming,
 skimming, slamming, slimming, slumming, spamming,
 stemming, strumming, summing, swimming, thrumming,
 tramming, trimming, unjamming, whamming, whimming}
```

Recall the regular expression `\\ w +` represents a word character repeated some number of times; `\\ b` represents a word boundary; the `$1` refers to the first expression `(\\ w +)`, that is, the characters up to the *mmming*. These characters will be joined with a single *m*.

```
rule2 = RegularExpression["(\\w+)mming\\b"] :→ "$1" ~~ "m";

StringReplace[words, rule2]

{bedim, brim, bum, chum, clam, cram, dam, deprogram, diagram,
 dim, drum, flimflam, gum, ham, hem, hum, jam, lam, lem,
 monogram, multiprogram, program, ram, reprogram, rim, scam,
 scram, scrum, scum, sham, shim, skim, slam, slim, slum, spam,
 stem, strum, sum, swim, thrum, tram, trim, unjam, wham, whim}
```

Again, this is quite good although the word *lemming* has been stemmed to the nonword *lem*, something that will need to be dealt with as a special case. The way to do that is to order the rules so that the special cases are caught first.

```
rule2 = {"lemming" :→ "lemming",
    RegularExpression["(\\w+)mming\\b"] :→ "$1" ~~ "m"
   };

StringReplace[words, rule2]

{bedim, brim, bum, chum, clam, cram, dam, deprogram,
 diagram, dim, drum, flimflam, gum, ham, hem, hum,
 jam, lam, lemming, monogram, multiprogram, program,
 ram, reprogram, rim, scam, scram, scrum, scum, sham,
 shim, skim, slam, slim, slum, spam, stem, strum,
 sum, swim, thrum, tram, trim, unjam, wham, whim}
```

■ Words Ending in ...otes

Numerous rules are needed for turning plural words into their singular stems. To see this, consider a naive rule that simply drops the *s* for any such words.

```
StringReplace[{"possess", "thrushes", "oasis"},
 RegularExpression["(\\w+)s"] :→ "$1"]

{posses, thrushe, oasi}
```

This is clearly too general a rule. In fact, several different rules are needed for words that end in *s*, depending upon the preceding characters. Here, we will only deal with words that end in …*otes*. First gather the words in the dictionary that match this pattern.

```
words =
 DictionaryLookup[
   w__ /; StringTake[w, -4] === "otes" &&
     LowerCaseQ@StringTake[w, 1]] // Quiet
```

```
{anecdotes, antidotes, asymptotes, banknotes,
 compotes, connotes, cotes, coyotes, creosotes,
 demotes, denotes, devotes, dotes, dovecotes,
 emotes, footnotes, garrotes, keynotes, litotes,
 misquotes, motes, notes, outvotes, promotes, quotes,
 remotes, rotes, totes, unquotes, votes, zygotes}
```

Here is the replacement rule.

```
rule3 = RegularExpression["(\\w+)(ote)s"] :→ "$1$2";
```

```
StringReplace[words, rule3]
```

```
{anecdote, antidote, asymptote, banknote, compote,
 connote, cote, coyote, creosote, demote, denote,
 devote, dote, dovecote, emote, footnote, garrote,
 keynote, litote, misquote, mote, note, outvote, promote,
 quote, remote, rote, tote, unquote, vote, zygote}
```

Stemming *litotes* gives the nonword *litote*. This can again be resolved by adding some specific rules for these not uncommon situations.

■ Plural to Singular

Let us try to deal with the general problem of stemming plural forms to singular. This is a more difficult scenario to deal with as there are many rules and even more exceptions. We will begin by showing how the order of the replacement rules matters in the stemming process.

You might imagine the rules given in Table 9.4 being used to stem plurals (these are not complete, but they will get us started). In fact, these are step 1a of the commonly-used Porter's algorithm for word stemming in the English language.

Rule	Example
…sses → *…ss*	posseses → possess
…shes → *…sh*	churches → church
…ies → *…y*	theories → theory
…ss → *…ss*	pass → pass
…us → *…us*	abacus → abacus
…s → *…*	cats → cat

▲ **Table 9.4.** Stemming rules, plural to singular

The order in which such rules are used is important. You do not want the last rule being used before any of the others. As we saw with the previous set of rules, *Mathematica* will apply rules in the order in which they are given, assuming that they have roughly the same level of specificity. Note also that some of these rules are designed to leave certain words unchanged. For example neither *pass* nor *abacus* are plural and they should not be stemmed.

Here then is a rough attempt at stemming plural words. First we gather the words from the dictionary that end in *s* and display a random sample of them.

```
words = DictionaryLookup[
   w__ /; StringTake[w, -1] == "s" &&
     LowerCaseQ@StringTake[w, 1]];
randwords = RandomSample[words, 30]

{infiltrates, cavers, rumpus, parliamentarians,
 whippoorwills, briefs, neoplasms, haggis, profess,
 inherits, colonials, impulsiveness, stipulations,
 glops, splays, observatories, stowaways, rapes,
 tumbles, minters, cauliflowers, annoyances,
 newsletters, steeplechases, bursitis, graphologists,
 spathes, steepens, shortcuts, explicates}
```

```
rules = {
   RegularExpression["(\\w+)(ss)(es)"] :> "$1$2",
   RegularExpression["(\\w+)(sh)(es)"] :> "$1$2",
   RegularExpression["(\\w+)(ies)"] :> "$1" ~~ "y",
   RegularExpression["(\\w+)(ss)"] :> "$1$2",
   RegularExpression["(\\w+)(us)"] :> "$1$2",
   RegularExpression["(\\w+)(s)"] :> "$1"
  };
```

```
StringReplace[randwords, rules]
```

```
{infiltrate, caver, rumpus, parliamentarian,
 whippoorwill, brief, neoplasm, haggi, profess,
 inherit, colonial, impulsiveness, stipulation, glop,
 splay, observatory, stowaway, rape, tumble, minter,
 cauliflower, annoyance, newsletter, steeplechase, bursiti,
 graphologist, spathe, steepen, shortcut, explicate}
```

This process of word stemming requires a lot of trial and error and the creation of many rules for the exceptions. Another approach, called *lemmatization*, does a more careful and thorough job by working with vocabularies and performing morphological analysis of the words to better understand how to reduce them to a root. For more information, see Manning, Raghavan, and Schütze (2008) [6].

■ *Exercises*

1. Rewrite the genomic example in Section 9.3 to use regular expressions instead of string patterns to find all occurrences of the sequence `AA`*anything*`T`. Here is the example using general string patterns.

```
gene = GenomeData["IGHV357"];
```

```
StringCases[gene, "AA" ~~ _ ~~ "T"]
```

```
{AAGT, AAGT, AAAT, AAGT, AAAT, AAAT}
```

2. Rewrite the web page example in Section 9.3 to use regular expressions to find all phone numbers on the page, that is, expressions of the form *nnn–nnn–nnnn*. Modify accordingly for other web pages and phone numbers formatted for other regions.

3. Create a function `UcLc[`*word*`]` that takes its argument *word* and returns the word with the first letter uppercase and the rest of the letters lowercase.

4. Use a regular expression to find all words given by `DictionaryLookup` that consist only of the letters *a*, *e*, *i*, *o*, *u*, and *y* in any order with any number of repetitions of the letters.

5. The basic rules for pluralizing words in the English language are roughly, as follows: if a noun ends in *ch*, *s*, *sh*, *j*, *x*, or *z*, it is made plural by adding *es* to the end. If the noun ends in *y* and is preceded by a consonant, replace the *y* with *ies*. If the word ends in *ium*, replace with *ia* (*Chicago Manual of Style* 2010 [9]). Of course, there are many more rules and even more exceptions, but you can implement a basic set of rules to convert singular words to plural based on these rules and then try them out on the following list of words.

```
words = {"building", "finch", "fix", "ratio", "envy",
   "boy", "baby", "faculty", "honorarium"};
```

6. A common task in transcribing audio is cleaning up text, removing certain phrases such as *um*, *er*, and so on, and other tags that are used to make a note of some sort. For example, the following transcription of a lecture from the University of Warwick, Centre for Applied Linguistics (BASE Corpus [10]), contains quite a few fragments that should be removed, including newline characters, parenthetical remarks, and nonwords. Use `StringReplace` with the appropriate rules to "clean" this text and then apply your code to a larger corpus.

```
text =
  "okay well er today we're er going to be carrying
    on with the er French \nRevolution you may
    have noticed i was sort of getting rather er
    enthusiastic \nand carried away at the end of
    the last one i was sort of almost er like i
    sort \nof started at the beginning about
    someone standing on a coffee table and s-,
    \nshouting to arms citizens as if i was going
    to sort of leap up on the desk and \nsay to
    arms let's storm the Rootes Social Building
    [laughter] or er let's go \nout arm in arm
    singing the Marseillaise or something er like that";
```

7. Find the distribution of sentence lengths for any given piece of text. `ExampleData["Text"]` contains several well-known books and documents that you can use. You will need to think about and identify sentence delimiters carefully. Take care to deal properly with words such as *Mr*., *Dr*., and so on that might incorrectly trigger a sentence-ending detector.

8. In web searches and certain problems in natural language processing (NLP), it is often useful to filter out certain words prior to performing the search or processing of the text to help with the performance of the algorithms. Words such as *the*, *and*, *is*, and so on are commonly referred to as *stop words* for this purpose. Lists of stop words are almost always created manually based on the constraints of a particular application. We will assume you can import a list of stop words as they are commonly available across the internet. For our purposes here, we will use one such list that comes with the materials for this book.

```
stopwords = Rest@Import["StopWords.dat", "List"];
RandomSample[stopwords, 12]
```

```
{what, look, taken, specify, wants, thorough,
 they, hello, whose, them, mightn't, particular}
```

Using the above list of stop words, or any other that you are interested in, first filter some sample "search phrases" and then remove all stop words from a larger piece of text.

```
searchPhrases = {"Find my favorite phone",
   "How deep is the ocean?",
   "What is the meaning of life?"};
```

9. Modify the previous exercise so that the user can supply a list of punctuation in addition to the list of stop words to be used to filter the text.

■ 9.5. Examples and Applications

This section puts together many of the concepts and techniques developed earlier in the chapter to solve several nontrivial applied problems. The first example creates a function to generate random strings, mirroring the syntax of the built-in random number functions. People who work with large strings, such as those in genomic research, often partition their strings into small blocks and then perform some analysis on those substrings. We develop functions for partitioning strings as well as several examples for analyzing sequences of genetic code. An additional example covers checksums, which are used to verify stored and transmitted data. Finally, a word game is included in which we create blanagrams, a variant of anagrams.

□ Random Strings

A blasphemous sect suggested …that all men should juggle letters and symbols until they constructed, by an improbable gift of chance, these canonical books.
— Jorge L. Borges, *The Library of Babel* [11]

Those who work with genomic data often need to test their algorithms on strings. While it may be sensible to test against real data – for example, using genes on the human genome – random data might be more appropriate to quickly test and measure the efficiency of an algorithm. Although *Mathematica* has a variety of functions for creating random numbers, random variates, and so on, it does not have a function to create random sequences of strings. In this section we will create one.

To start, we will choose the characters A, C, T, and G – representing the nucleotide, or DNA, bases – as our alphabet, that is, the letters in the random strings we will create.

```
chars = {"A", "C", "T", "G"};
```

The key observation is that we want to choose one character at random from this list. Since we need to repeat this *n* times, we need to randomly choose with replacement. That is the purpose of `RandomChoice`.

```
RandomChoice[chars, 10]
```

```
{T, T, C, T, T, G, T, G, A, T}
```

This expression is a list of strings.

```
FullForm[%]
```

```
List["T", "T", "C", "T", "T", "G", "T", "G", "A", "T"]
```

Finally, we concatenate the strings.

```
StringJoin[%] // FullForm
```

```
"TTCTTGTGAT"
```

So a first attempt at putting these pieces together would look like this. Note the use of a default value of 1 for the optional argument n (see Section 6.1 for a discussion of default values for arguments to functions).

```
RandomString[chars_List, n_Integer: 1] :=
 StringJoin[RandomChoice[chars, n]]
```

```
RandomString[{"A", "C", "T", "G"}, 500]
```

```
ACGCTCGGGGTATTGAGCGTGTTTCCTCGTGCGTTCGGCTTAGGATTAGGCGGGACCGT⋱
    AGTACTACGTTTAATATGTTCTTCACGAGGTTTATATTTTCCTAAAGACGCCCTC⋱
    ATCCAACTGTCCAGTTGCCCAACTTCGGTTCGTATGTATTTAATAGCCTTGTGAT⋱
    CAAATTGGGTCACGTATCCGCCTAATGGTTTTTTTGCGTATAAGCCACCTTCGCA⋱
    CATCTAGGGGTGATAAGGGAACTATCATAAATAGCTGACACCCTCGCCATACGAC⋱
    TTTATTTGGCACAATGTTTGGTCTCGTTTAGACGAAACCAAAGGGTTGCTATCGG⋱
    TATAACTGGGCGACAACGACAACAATTGCCATAAGGTCAGGTAGGACCACTTTAT⋱
    ATCCCTTGCCATCGGTATACATCTTCACTGGACGTTACGACTGTACTAAGAGTTG⋱
    GGCCCAACTAATCGCGAGATAAAACCTTTCCTGTAGTTTTTCAAGCTCGTGCATT⋱
    C
```

The default value of the second argument gives one choice.

```
RandomString[{"A", "C", "T", "G"}]
```

```
A
```

We can make the arguments a bit more general using structured patterns. The first argument in this next version must be a list consisting of a sequence of one or more strings.

```
Clear[RandomString]
```

```
RandomString[{ch__String}, n_Integer: 1] :=
 StringJoin[RandomChoice[{ch}, n]]
```

```
RandomString[{"a", "b", "d"}, 12]
```

```
aadbbddbddbd
```

Here is a ten-character password generator.

```
RandomString[CharacterRange["A", "z"] ⋃
  CharacterRange["0", "9"], 10]
```

```
kRP^Rqih`B
```

It is not hard to extend this function to create *n* random strings of a given length. We essentially pass that argument structure to `RandomChoice`.

```
RandomString[{ch__String}, {n_Integer, len_Integer}] :=
 Map[StringJoin, RandomChoice[{ch}, {n, len}]]
```

```
RandomString[{"A", "C", "T", "G"}, {4, 12}]
```

```
{TGCTATGCTCCT, GGTATGTCCCTA, AGACCAAGACAT, CTCCTCAAGCAT}
```

```
RandomString[{"A", "C", "T", "G"}, {50, 1}]
```

```
{C, T, C, A, A, C, G, C, T, A, C, G, T, G, C, C,
 C, A, T, C, G, C, G, G, A, C, G, G, C, G, A, C, G,
 A, A, A, C, G, C, C, G, A, G, T, G, G, G, G, A, T}
```

The exercises at the end of this section include a problem that asks you to add an option that provides a mechanism to weight the individual characters in the random string.

□ Partitioning Strings

Some string analysis requires strings to be broken up into blocks of a certain size and then computations are performed on those blocks. Although there is no built-in function for partitioning strings, we can easily create one, taking advantage of the syntax and speed of the built-in `Partition` function.

The `Partition` function requires a list as its first argument. To start, we will give it a list of the characters in a prototype string, a gene on the human genome.

```
GenomeData["IGHVII671", "Name"]
```

```
immunoglobulin heavy variable (II)-67-1
```

```
str = GenomeData["IGHVII671"]

ATGTCCTATTCAGGAGCAGCTACAGCAGTCATGCCTAGGTGTGAAGATCACACACTGAC
   CTCACCCATGCTGTCTCTGGCCACTTCATCACAACCAATGCTTAATATTGGACGT
   GGATCTGCCAGTCCCCGGGGAATGGGTTGAATGGAT
```

```
Characters[str]

{A, T, G, T, C, C, T, A, T, T, C, A, G, G, A, G, C, A,
 G, C, T, A, C, A, G, C, A, G, T, C, A, T, G, C, C, T, A,
 G, G, T, G, T, G, A, A, G, A, T, C, A, C, A, C, A, C,
 T, G, A, C, C, T, C, A, C, C, C, A, T, G, C, T, G, T, C,
 T, C, T, G, G, C, C, A, C, T, T, C, A, T, C, A, C, A, A,
 C, C, A, A, T, G, C, T, T, A, A, T, A, T, T, G, G, A, C,
 G, T, G, G, A, T, C, T, G, C, C, A, G, T, C, C, C, C, G,
 G, G, G, A, A, T, G, G, G, T, T, G, A, A, T, G, G, A, T}
```

Now, partition this list of characters into lists of length 4 with offset 1.

```
Partition[Characters[str], 4, 4, 1]

{{A, T, G, T}, {C, C, T, A}, {T, T, C, A},
 {G, G, A, G}, {C, A, G, C}, {T, A, C, A},
 {G, C, A, G}, {T, C, A, T}, {G, C, C, T}, {A, G, G, T},
 {G, T, G, A}, {A, G, A, T}, {C, A, C, A}, {C, A, C, T},
 {G, A, C, C}, {T, C, A, C}, {C, C, A, T}, {G, C, T, G},
 {T, C, T, C}, {T, G, G, C}, {C, A, C, T}, {T, C, A, T},
 {C, A, C, A}, {A, C, C, A}, {A, T, G, C}, {T, T, A, A},
 {T, A, T, T}, {G, G, A, C}, {G, T, G, G}, {A, T, C, T},
 {G, C, C, A}, {G, T, C, C}, {C, C, G, G}, {G, G, A, A},
 {T, G, G, G}, {T, T, G, A}, {A, T, G, G}, {A, T, A, T}}
```

Because the number of characters in `str` is not a multiple of 4, this use of `Partition` has padded the last sublist with the first two characters from the original string; in other words, this has treated the list cyclically; not quite what we want here.

```
Mod[StringLength[str], 4] == 0

False
```

A slightly different syntax for `Partition` gives an uneven subset at the end. We will need to use this form so as not to lose or introduce any spurious information.

```
parts = Partition[Characters[str], 4, 4, 1, {}]

{{A, T, G, T}, {C, C, T, A}, {T, T, C, A},
 {G, G, A, G}, {C, A, G, C}, {T, A, C, A},
 {G, C, A, G}, {T, C, A, T}, {G, C, C, T}, {A, G, G, T},
 {G, T, G, A}, {A, G, A, T}, {C, A, C, A}, {C, A, C, T},
 {G, A, C, C}, {T, C, A, C}, {C, C, A, T}, {G, C, T, G},
 {T, C, T, C}, {T, G, G, C}, {C, A, C, T}, {T, C, A, T},
 {C, A, C, A}, {A, C, C, A}, {A, T, G, C}, {T, T, A, A},
 {T, A, T, T}, {G, G, A, C}, {G, T, G, G}, {A, T, C, T},
 {G, C, C, A}, {G, T, C, C}, {C, C, G, G}, {G, G, A, A},
 {T, G, G, G}, {T, T, G, A}, {A, T, G, G}, {A, T}}
```

Finally, convert each sublist into a contiguous string.

```
Map[StringJoin, parts]

{ATGT, CCTA, TTCA, GGAG, CAGC, TACA, GCAG, TCAT, GCCT, AGGT,
 GTGA, AGAT, CACA, CACT, GACC, TCAC, CCAT, GCTG, TCTC,
 TGGC, CACT, TCAT, CACA, ACCA, ATGC, TTAA, TATT, GGAC,
 GTGG, ATCT, GCCA, GTCC, CCGG, GGAA, TGGG, TTGA, ATGG, AT}
```

This puts everything together in a function.

```
StringPartition[str_String, blocksize_] :=
 Map[StringJoin, Partition[Characters[str], blocksize,
   blocksize, 1, {}]]
```

This partitions the string into nonoverlapping blocks of length 12.

```
StringPartition[str, 12]

{ATGTCCTATTCA, GGAGCAGCTACA, GCAGTCATGCCT,
 AGGTGTGAAGAT, CACACACTGACC, TCACCCATGCTG,
 TCTCTGGCCACT, TCATCACAACCA, ATGCTTAATATT,
 GGACGTGGATCT, GCCAGTCCCCGG, GGAATGGGTTGA, ATGGAT}
```

This function operates on large strings fairly fast. Here we partition a random string of length ten million into nonoverlapping blocks of length ten.

```
data = RandomString[{"A", "T", "C", "G"}, 10^7];

  Timing[StringPartition[data, 10];]

{4.049500, Null}
```

□ Adler Checksum

Checksums, or hashes, are commonly used to check the integrity of data when that data are either stored or transmitted. A checksum might be created, for example, when some data are stored on a disk. To check the integrity of that data, the checksum can be recomputed and if it differs from the stored value, there is a very high probability that the data was tampered with. Hash functions are used to create hash tables which are used for record lookup in large arrays of data. As an example of the use of character codes, we will implement a basic checksum algorithm, the Adler checksum.

Mathematica has a built-in function, `Hash`, that can be used to create hash codes, or checksums.

```
Hash["Mathematica"]
```

```
7 984 226 549 216 812 226
```

If the string is changed, its checksum changes accordingly.

```
Hash["mathematica"]
```

```
1 307 656 555 132 665 570
```

We will implement a basic hash code, known as the Adler-32 checksum algorithm. Given a string $c_1 c_2 \cdots c_n$ consisting of concatenated characters c_i, we form two 16-bit sums m and n as follows:

$$m = 1 + \mathrm{cc}_1 + \mathrm{cc}_2 + \cdots + \mathrm{cc}_n \bmod 65\,521,$$
$$n = (1 + \mathrm{cc}_1) + (1 + \mathrm{cc}_1 + \mathrm{cc}_2) + \cdots + (1 + \mathrm{cc}_1 + \mathrm{cc}_2 + \cdots + \mathrm{cc}_n) \bmod 65\,521,$$

where cc_i is the character code for the character c_i. The number 65521 is chosen as it is the largest prime smaller than 2^{16}. Choosing primes for this task seems to reduce the probability that an interchange of two bytes will not be detected. Finally, the Adler checksum is given by

$$m + 65\,536\,n$$

Let us take *Mathematica* as our test word. We start by getting the Ascii character codes for each character.

```
str = "Mathematica";
codes = ToCharacterCode[str]
```

```
{77, 97, 116, 104, 101, 109, 97, 116, 105, 99, 97}
```

The number *m* above is given by the cumulative sums of the character codes, with 1 prepended to that list. (This step could also be done using `FoldList`.)

```
mList = Accumulate[Join[{1}, codes]]

{1, 78, 175, 291, 395, 496, 605, 702, 818, 923, 1022, 1119}

m = Last[mList]

1119
```

The number *n* is given by the cumulative sums from this last list, omitting the 1 at the beginning as it is already part of the cumulative sums.

```
nList = Accumulate[Rest[mList]]

{78, 253, 544, 939, 1435, 2040, 2742, 3560, 4483, 5505, 6624}

n = Last[nList]

6624

m + 65 536 n

434 111 583
```

We can check our result against the algorithm implemented in the `Hash` function.

```
Hash["Mathematica", "Adler32"]

434 111 583
```

Finally, this puts these steps together in a reusable function. Our prototype worked with small numbers and so the need to work mod 65521 was not necessary. For general inputs, the arithmetic will be done using this modulus.

```
AdlerChecksum[str_String] := Module[{codes, n, m},
  codes = ToCharacterCode[str];
  m = Mod[Accumulate[Join[{1}, codes]], 65 521];
  n = Mod[Accumulate[Rest[m]], 65 521];
  Last[m] + Last[n] 65 536
 ]

AdlerChecksum["Mathematica"]

434 111 583
```

As an aside, here is its hash code in hexadecimal.

```
IntegerString[%, 16]
```

```
19e0045f
```

And here is a lengthier example.

```
AdlerChecksum[
 "Lorem ipsum dolor sit amet, consectetur adipiscing
   elit. Fusce ultrices ornare odio. Proin adipiscing,
   mi non pharetra eleifend, nibh libero laoreet
   metus, at imperdiet urna ante in lectus."]
```

```
3 747 169 622
```

□ Search for Substrings

As we have seen in this chapter, string patterns provide a powerful and compact mechanism for operating on text data. In this example, we will create a function that searches the dictionary for words containing a specified substring.

If our test substring is *cite*, here is how we would find all words that end in *cite*. Note the triple blank pattern to match any sequence of zero or more characters.

```
DictionaryLookup[___ ~~ "cite"]
```

```
{anthracite, calcite, cite, excite,
 incite, Lucite, overexcite, plebiscite, recite}
```

Here are all words that begin with *cite*.

```
DictionaryLookup["cite" ~~ ___]
```

```
{cite, cited, cites}
```

And this gives all words that have *cite* somewhere in them, at the beginning, middle, or end.

```
DictionaryLookup[___ ~~ "cite" ~~ ___]
```

```
{anthracite, calcite, cite, cited, cites, elicited,
 excite, excited, excitedly, excitement, excitements,
 exciter, exciters, excites, incite, incited, incitement,
 incitements, inciter, inciters, incites, Lucite,
 Lucites, overexcite, overexcited, overexcites,
 plebiscite, plebiscites, recite, recited, reciter,
 reciters, recites, solicited, unexcited, unsolicited}
```

Using the double blank gives words that have *cite* in them but not beginning or ending with *cite*.

```
DictionaryLookup[__ ~~ "cite" ~~ __]
```

```
{elicited, excited, excitedly, excitement, excitements,
 exciter, exciters, excites, incited, incitement,
 incitements, inciter, inciters, incites, Lucites,
 overexcited, overexcites, plebiscites, recited, reciter,
 reciters, recites, solicited, unexcited, unsolicited}
```

Let us put these pieces together in a reusable function `FindWordsContaining`. We will include one option, `WordPosition` that identifies where in the word the substring is expected to occur.

```
Options[FindWordsContaining] = {WordPosition → "Start"};
```

Depending upon the value of the option `WordPosition`, `Which` directs which expression will be evaluated.

```
FindWordsContaining[str_String, OptionsPattern[]] :=
 Module[{wp = OptionValue[WordPosition]},
  Which[
   wp == "Start", DictionaryLookup[str ~~ ___],
   wp == "Middle", DictionaryLookup[__ ~~ str ~~ __],
   wp == "End", DictionaryLookup[___ ~~ str],
   wp == "Anywhere", DictionaryLookup[___ ~~ str ~~ ___]
  ]]
```

Using the default value for `WordPosition`, this finds all words in the dictionary that start with the string *cite*.

```
FindWordsContaining["cite"]
```

```
{cite, cited, cites}
```

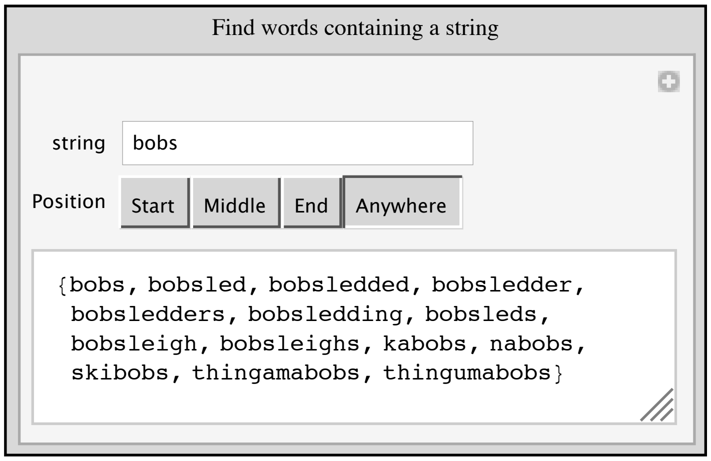
And this finds all words that have *cite* anywhere in the word.

```
FindWordsContaining["cite", WordPosition → "Anywhere"]
```

```
{anthracite, calcite, cite, cited, cites, elicited,
 excite, excited, excitedly, excitement, excitements,
 exciter, exciters, excites, incite, incited, incitement,
 incitements, inciter, inciters, incites, Lucite,
 Lucites, overexcite, overexcited, overexcites,
 plebiscite, plebiscites, recite, recited, reciter,
 reciters, recites, solicited, unexcited, unsolicited}
```

Finally, here is a dynamic interface that includes a text field in which you can enter an input string; tabs are used to specify in which part of the word you expect the string to occur.

```
Framed[
 Labeled[
  Manipulate[FindWordsContaining[ToString@string,
    WordPosition → pos],
   {string, bobs}, {{pos, "Anywhere", "Position"},
    {"Start", "Middle", "End", "Anywhere"}},
   ContentSize → {300, 80}, SaveDefinitions → True],
  Text@"Find words containing a string", Top],
 Background → LightGray]
```


Find words containing a string

string: bobs

Position: Start | Middle | End | [✓ Anywhere]

```
{bobs, bobsled, bobsledded, bobsledder,
 bobsledders, bobsledding, bobsleds,
 bobsleigh, bobsleighs, kabobs, nabobs,
 skibobs, thingamabobs, thingumabobs}
```

For more on the creation of these sorts of dynamic interfaces, see Chapter 11.

□ DNA Sequence Analysis

DNA molecules are composed of sequences of the nitrogenous bases guanine, cytosine, thymine, and adenine. Guanine and cytosine bond with three hydrogen bonds and thymine and adenine bond with two. Research has indicated that high GC content (guanine and cytosine) DNA is more stable than that with lower GC. The exact reasons for this are not completely understood and determining the GC content of various DNA materials is an active area of biomolecular research. GC content is often described as a percentage of the guanine and cytosine nucleotides compared to the entire nucleotide content (Cristianini and Hahn 2007 [12]). In this section we will create a function to compute the ratio of GC in any given DNA sequence or fragment.

We will start by importing a FASTA file consisting of human mitochondrial DNA, displaying some information about the contents of this file.

```
hsMito = Import["ExampleData/mitochondrion.fa.gz"];
```

```
Import["ExampleData/mitochondrion.fa.gz",
 {"FASTA", "Header"}]
```

```
{gi|17981852|ref|NC_001807.4|
   Homo sapiens mitochondrion, complete genome}
```

```
StringLength[hsMito]
```

```
{16 571}
```

```
StringTake[hsMito, 500]
```

```
{GATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATT⋱
    TTCGTCTGGGGGGTGTGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCA⋱
    CCCTATGTCGCAGTATCTGTCTTTGATTCCTGCCTCATTCTATTATTTATCGC⋱
    ACCTACGTTCAATATTACAGGCGAACATACCTACTAAAGTGTGTTAATTAATT⋱
    AATGCTTGTAGGACATAATAATAACAATTGAATGTCTGCACAGCCGCTTTCCA⋱
    CACAGACATCATAACAAAAAATTTCCACCAAACCCCCCCCTCCCCCCGCTTCT⋱
    GGCCACAGCACTTAAACACATCTCTGCCAAACCCCAAAAACAAAGAACCCTAA⋱
    CACCAGCCTAACCAGATTTCAAATTTTATCTTTAGGCGGTATGCACTTTTAAC⋱
    AGTCACCCCCCAACTAACACATTATTTTCCCCTCCCACTCCCATACTACTAAT⋱
    CTCATCAATACAACCCCC}
```

We use `StringCount` to count the number of occurrences of G or C in this sequence.

```
gc = StringCount[hsMito, "G" | "C"]
```

```
{7372}
```

And here is the number of occurrences of A or T.

```
at = StringCount[hsMito, "A" | "T"]

{9199}
```

The GC percentage is given by the following ratio.

$$N\left[\frac{gc}{gc + at}\right]$$

```
{0.444874}
```

Here then is an auxiliary function we will use in what follows.

```
gcRatio[ls_String] := Module[{gc, at},
  gc = StringCount[ls, "G" | "C"];
  at = StringCount[ls, "A" | "T"];
  N[gc / (gc + at)]
 ]
```

Note that `gcRatio` expects a string as an argument, but this fails with `hsMito`, imported from an external source.

```
Short[gcRatio[hsMito], 8]

gcRatio[
 {GATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTAT⋱
      TTTCGTCTGGGGGGTGTGCACGCGATAGCATTGCGAGACGCTGGAGCCG⋱
      GAGCACCCTATGTCGCAGTATCTGTCTTTGATTCCTGCCTCATTCTATT⋱
      ATTTATCGCACCTACGTTCAATATTACAGGCGAACATACCTACTAAAGT⋱
      GTGTTAATTAATTAATGCTTG …
   AGTCAAATCCCTTCTCGTCCCCATGGATGACCCCCCTCAGATAGGGGTCCCTTGA⋱
      CCACCATCCTCCGTGAAATCAATATCCCGCACAAGAGTGCTACTCTCCT⋱
      CGCTCCGGGCCCATAACACTTGGGGGTAGCTAAAGTGAACTGTATCCGA⋱
      CATCTGGTTCCTACTTCAGGGCCATAAAGCCTAAATAGCCCACACGTTC⋱
      CCCTTAAATAAGACATCACGATG}]
```

It fails because `Import` returns a list consisting of a string, not a raw string. We can remedy this by writing a rule to deal with this argument structure and then call the first rule.

```
gcRatio[{str_String}] := gcRatio[str]

gcRatio[hsMito]

0.444874
```

Typically, researchers are interested in studying the GC ratio on particular fragments of DNA and comparing it with similar fragments on another molecule. One common way of doing this is to compute the GC ratio for blocks of nucleotides of some given length. We will use the function `StringPartition`, developed earlier to partition the sequence into blocks of a given size. We will work with a small random sequence to prototype.

```
blocksize = 10;
str = RandomString[{"A", "C", "T", "G"}, 125];
lis = StringPartition[str, blocksize]

{CCTTCCACAA, ACAGTCGCGG, TTATCTTGTC, ACAACGAGAC,
 AGCGCAGAGT, CCTGGGGGTG, CGCAAGCTTC, CGCAGAAACT,
 AGTCCAGGCA, ACTATCTCCG, GTAAGTTTGC, TGCTCTCCCC, GATAT}
```

Here are the GC ratios for each of the blocks given by `lis`.

```
Map[gcRatio, lis]

{0.5, 0.7, 0.3, 0.5, 0.6, 0.8,
 0.6, 0.5, 0.6, 0.5, 0.4, 0.7, 0.2}
```

Finally, it is helpful to be able to identify each block by its starting position. So we first create a list of the starting positions for each block and then transpose that with the ratios.

```
Table[i, {i, 1, StringLength[str], blocksize}]

{1, 11, 21, 31, 41, 51, 61, 71, 81, 91, 101, 111, 121}
```

```
Transpose[{Table[i, {i, 1, StringLength[str], blocksize}],
  Map[gcRatio, lis]}]

{{1, 0.5}, {11, 0.7}, {21, 0.3}, {31, 0.5},
 {41, 0.6}, {51, 0.8}, {61, 0.6}, {71, 0.5}, {81, 0.6},
 {91, 0.5}, {101, 0.4}, {111, 0.7}, {121, 0.2}}
```

Here are all the pieces in one function, `GCRatio`.

```
GCRatio[str_String, blocksize_Integer] :=
 Module[{lis, blocks},
  lis = StringPartition[str, blocksize];
  blocks = Table[i, {i, 1, StringLength[str], blocksize}];
  Transpose[{blocks, Map[gcRatio, lis]}]
 ]
```

And again, a second rule in case the string is wrapped in a list.

```
GCRatio[{str_String}, blocksize_Integer] :=
 GCRatio[str, blocksize]
```

Let us try it out first on our prototype sequence.

```
GCRatio[str, 10]
```

```
{{1, 0.5}, {11, 0.7}, {21, 0.3}, {31, 0.5},
 {41, 0.6}, {51, 0.8}, {61, 0.6}, {71, 0.5}, {81, 0.6},
 {91, 0.5}, {101, 0.4}, {111, 0.7}, {121, 0.2}}
```

And then on the human mitochondrial DNA with block size 1000.

```
gcdata = GCRatio[hsMito, 1000]
```

```
{{1, 0.46}, {1001, 0.441}, {2001, 0.43}, {3001, 0.478},
 {4001, 0.427}, {5001, 0.439}, {6001, 0.474}, {7001, 0.438},
 {8001, 0.434}, {9001, 0.461}, {10 001, 0.396},
 {11 001, 0.448}, {12 001, 0.429}, {13 001, 0.471},
 {14 001, 0.435}, {15 001, 0.446}, {16 001, 0.464098}}
```

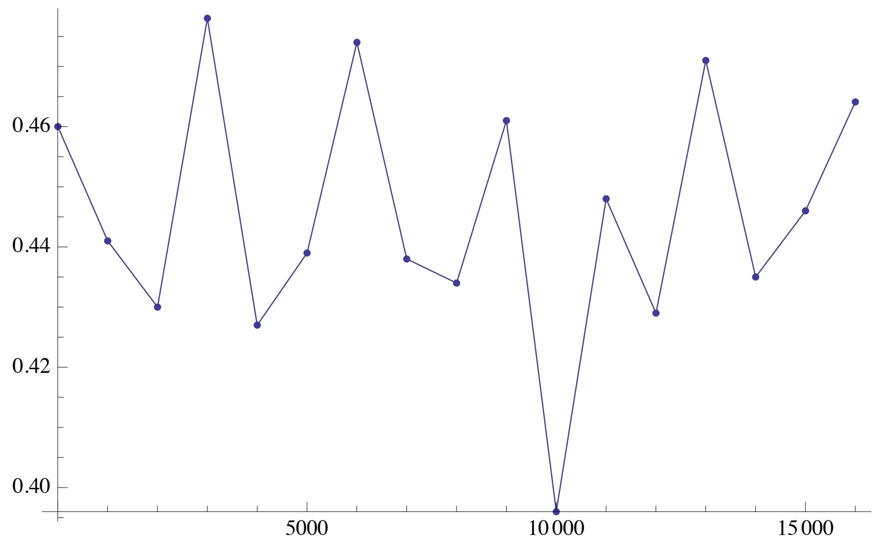
Various types of analysis can then be performed on these blocks. For example, using `Select`, this quickly finds regions of high GC content.

```
Select[gcdata, Last[#] > 0.47 &]
```

```
{{3001, 0.478}, {6001, 0.474}, {13 001, 0.471}}
```

Here is a quick visualization of the GC content across the blocks.

```
ListLinePlot[gcdata, Mesh → All]
```



Numerous comparative studies have been done looking at the GC content for different organisms. One much-studied organism is *Thermoplasma volcanium*, a bacterium-like organism that exists in very high-acid and high-temperature environments. To accommodate the extreme conditions, organisms in such environments often have high GC content which has a higher thermal stability. The following sequence is in the public domain and was obtained courtesy of the National Center for Biotechnology Information (NCBI Nucleotide Database [13]).

```
thermoVolc = Import["638154522.tar.gz",
   "638154522/638154522.fna"];
StringTake[thermoVolc, 250]
```

```
{TTTGTATAAGAAAAAATAGGAAAGGTTAATATCCATGCTCATATGGCTGTCCGAAAAA⋱
    ATCAATAACGAATATTAACCACGATAAAATAAGGTAAGGAAAGAATCCTGCAT⋱
    GAGCACAATAGAAGAACGCATTAAGGAAATAGAAGACGAAATCAAGAGAACTC⋱
    AGTACAATAAAGCCACTGAACACCACATCGGGCTTCTAAAAGCCAAGATTGCA⋱
    AGGCTCCAGATGGAGGCTAGAGCCCATAAAGGA}
```

```
StringLength[First@thermoVolc]
```

```
1 584 804
```

Here is the GC ratio for the entire sequence.

```
gcRatio[thermoVolc]
```

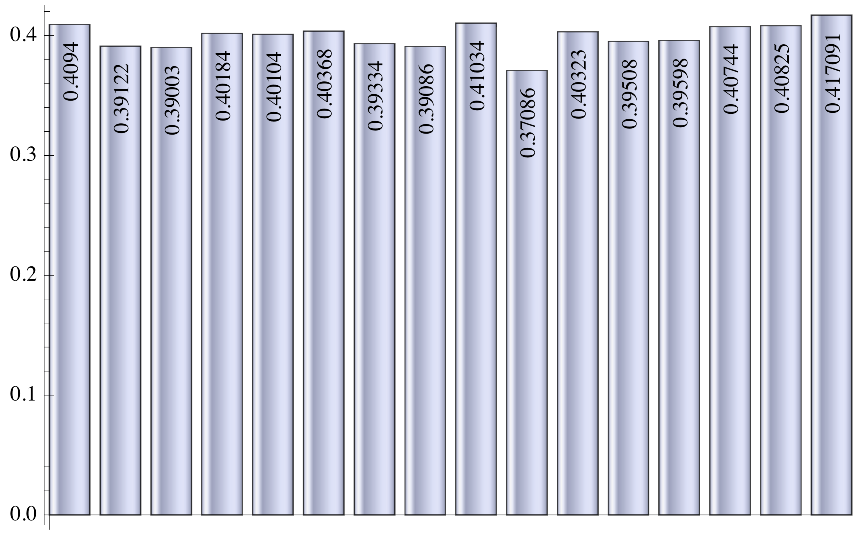
```
0.399185
```

And here are the ratios for block sizes of 100000.

```
tVratios = GCRatio[thermoVolc, 10^5]
```

```
{{1, 0.4094}, {100 001, 0.39122}, {200 001, 0.39003},
 {300 001, 0.40184}, {400 001, 0.40104}, {500 001, 0.40368},
 {600 001, 0.39334}, {700 001, 0.39086}, {800 001, 0.41034},
 {900 001, 0.37086}, {1 000 001, 0.40323}, {1 100 001, 0.39508},
 {1 200 001, 0.39598}, {1 300 001, 0.40744},
 {1 400 001, 0.40825}, {1 500 001, 0.417091}}
```

```
BarChart[Map[Last, tVratios], BarSpacing → Medium,
 ChartElementFunction → "GlassRectangle",
 ChartLabels → Placed[Map[Last, tVratios], Top,
   (Rotate[#, 90 Degree] &)]]
```



□ Displaying DNA Sequences

DNA sequences are typically long strings of nucleotides that are difficult to visualize simply by looking at the string of characters. Various visualization tools have been used to work with these sequences and in this section we will look at a common way of viewing them in a formatted table.

As before, we will prototype with a short random string consisting of nucleotide characters G, C, A, and T.

```
str = RandomString[{"G", "C", "A", "T"}, 125]
```

```
AATAGGATAATTGATAAATACTGCCGTAGCTACTATCGGCTATGAATTTATCGAAAGTT⋱
   TTCGCCCCGAGATACTAGATCTCGTACATTTATCGAGCTGTACCCGCAAACATGT⋱
   ATGTAACTTCC
```

Using `StringPartition` developed earlier in this chapter, we split the string into blocks of a desired size.

```
str1 = StringPartition[str, 10]
```

```
{AATAGGATAA, TTGATAAATA, CTGCCGTAGC, TACTATCGGC,
 TATGAATTTA, TCGAAAGTTT, TCGCCCCGAG, ATACTAGATC,
 TCGTACATTT, ATCGAGCTGT, ACCCGCAAAC, ATGTATGTAA, CTTCC}
```

We have 13 blocks here, but for readability purposes, we will put five blocks on each line of output. We use the blank string " " to pad out any string shorter than the blocksize, in this case 10.

```
str2 = Partition[str1, 5, 5, 1, " "]
```

```
{{AATAGGATAA, TTGATAAATA,
  CTGCCGTAGC, TACTATCGGC, TATGAATTTA},
 {TCGAAAGTTT, TCGCCCCGAG, ATACTAGATC, TCGTACATTT,
  ATCGAGCTGT}, {ACCCGCAAAC, ATGTATGTAA, CTTCC,  ,  }}
```

The following code gives the starting positions for each line once we have set the block length and row length.

```
blocklength = 10;
rowlength = 5;
ind = Select[Range[StringLength[str]],
  Mod[#, rowlength * blocklength] == 1 &]
```

```
{1, 51, 101}
```

We prepend the starting position of each row at the head of the row. Recall, the second argument to `Prepend` is the expression you wish to put in front (the indices) of your target expression (the rows).

```
MapThread[Prepend[#1, #2] &, {str2, ind}]

{{1, AATAGGATAA, TTGATAAATA, CTGCCGTAGC,
  TACTATCGGC, TATGAATTTA}, {51, TCGAAAGTTT,
  TCGCCCCGAG, ATACTAGATC, TCGTACATTT, ATCGAGCTGT},
 {101, ACCCGCAAAC, ATGTATGTAA, CTTCC,  ,  }}
```

This is what the formatted output should look like.

```
Grid[%, Alignment → {{Right, {Left}}, Automatic}]

  1 AATAGGATAA TTGATAAATA CTGCCGTAGC TACTATCGGC TATGAATTTA
 51 TCGAAAGTTT TCGCCCCGAG ATACTAGATC TCGTACATTT ATCGAGCTGT
101 ACCCGCAAAC ATGTATGTAA CTTCC
```

Finally, let us put this all together, setting up an option, `BlockSize` that is combined with the inherited options from `Grid`.

```
Options[SequenceTable] = Join[{BlockSize → 10}, Options[Grid]]

{BlockSize → 10, Alignment → {Center, Baseline},
 AllowedDimensions → Automatic, AllowScriptLevelChange → True,
 AutoDelete → False, Background → None,
 BaselinePosition → Automatic, BaseStyle → {},
 DefaultBaseStyle → Grid, DefaultElement → □,
 DeleteWithContents → True, Dividers → None,
 Editable → Automatic, Frame → None, FrameStyle → Automatic,
 ItemSize → Automatic, ItemStyle → None,
 Selectable → Automatic, Spacings → Automatic}
```

```
SequenceTable[lis_String, opts : OptionsPattern[]] :=
 Module[{n = OptionValue[BlockSize], len = StringLength[lis],
    rowlength = 5, str, blocks, ind},
   str = StringPartition[lis, n];
   blocks = Partition[str, 5, 5, 1, " "];
   ind = Select[Range[len], Mod[#, rowlength * n] == 1 &];
   Grid[MapThread[Prepend[#1, #2] &, {blocks, ind}],
    FilterRules[{opts}, Options[Grid]],
    Alignment → {{Right, {Left}}, Automatic}, Frame → True,
    Dividers → {{True, False}, All}]
 ]
```

```
str = RandomString[{"C", "A", "T", "G"}, 178]
```

```
ATCGACCTGCCTATGCGGGGCAATGCTATCTCGATCCTAGCGAGGCCCCTTACCGCGTG⋱
    GCCCTAGCATGGGCAGTAGTCTGTCGATAGCGGTTGCCAGCGCGTCAACTGGTTC⋱
    TGACGAGCCGACTCTCTACAGGTGCTAACTAGCCGACAGTGAATTCGCGCTTAGT⋱
    AACAGTGGT
```

```
SequenceTable[str]
```

```
  1  ATCGACCT⋱   CTATGCGG⋱   CAATGCTA⋱   TCGATCCT⋱   CGAGGCCC⋱

 51  TACCGCGT⋱   CCCTAGCA⋱   GGCAGTAG⋱   TGTCGATA⋱   GGTTGCCA⋱

101  GCGTCAAC⋱   GTTCTGAC⋱   GCCGACTC⋱   TACAGGTG⋱   AACTAGCC⋱

151  CAGTGAAT⋱   GCGCTTAG⋱   ACAGTGGT
```

Let us exercise some of the options.

```
SequenceTable[str, BlockSize → 12, Background → LightYellow,
 BaseStyle → Directive[FontSize → 8]]
```

```
  1  ATCGACCTGCCT  ATGCGGGGCAAT  GCTATCTCGATC  CTAGCGAGGCCC  CTTACCGCGTGG
 61  CCCTAGCATGGG  CAGTAGTCTGTC  GATAGCGGTTGC  CAGCGCGTCAAC  TGGTTCTGACGA
121  GCCGACTCTCTA  CAGGTGCTAACT  AGCCGACAGTGA  ATTCGCGCTTAG  TAACAGTGGT
```

□ Blanagrams

A blanagram is an anagram for another word except for the substitution of one letter. Think of Scrabble with a blank square (blank + anagram = blanagram). For example, *phyla* is a blanagram of *glyph*: replace the *g* with an *a* and find anagrams. In this section we will create a function that finds all blanagrams of a given word.

We will prototype with a simple word, *glyph*.

```
Characters["glyph"]
```

```
{g, l, y, p, h}
```

Start by replacing the first letter in *glyph* with an *a* and then finding all anagrams (using `Anagrams` from Section 9.2). The third argument to `StringReplacePart` is a list of beginning and ending positions for the replacement.

```
StringReplacePart["glyph", "a", {1, 1}]
```

```
alyph
```

```
Anagrams[%]
```

```
{phyla, haply}
```

Now do the same for each character position in the word.

```
Map[StringReplacePart["glyph", "a", {#, #}] &,
 Range[StringLength["glyph"]]]
```

```
{alyph, gayph, glaph, glyah, glypa}
```

Running `Anagrams` on each of these strings, only two appear as words in the dictionary.

```
Flatten[Map[Anagrams, %]]
```

```
{phyla, haply}
```

Having done this for the letter *a*, we now repeat for all other single characters.

```
CharacterRange["a", "z"]
```

```
{a, b, c, d, e, f, g, h, i, j, k,
 l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

```
blana = Table[
  Map[StringReplacePart["glyph", ch, {#, #}] &,
   Range[StringLength["glyph"]]],
  {ch, CharacterRange["a", "z"]}]
```

```
{{alyph, gayph, glaph, glyah, glypa},
 {blyph, gbyph, glbph, glybh, glypb},
 {clyph, gcyph, glcph, glych, glypc},
 {dlyph, gdyph, gldph, glydh, glypd},
 {elyph, geyph, gleph, glyeh, glype},
 {flyph, gfyph, glfph, glyfh, glypf},
 {glyph, ggyph, glgph, glygh, glypg},
 {hlyph, ghyph, glhph, glyhh, glyph},
 {ilyph, giyph, gliph, glyih, glypi},
 {jlyph, gjyph, gljph, glyjh, glypj},
 {klyph, gkyph, glkph, glykh, glypk},
 {llyph, glyph, gllph, glylh, glypl},
 {mlyph, gmyph, glmph, glymh, glypm},
 {nlyph, gnyph, glnph, glynh, glypn},
 {olyph, goyph, gloph, glyoh, glypo},
 {plyph, gpyph, glpph, glyph, glypp},
 {qlyph, gqyph, glqph, glyqh, glypq},
 {rlyph, gryph, glrph, glyrh, glypr},
 {slyph, gsyph, glsph, glysh, glyps},
 {tlyph, gtyph, gltph, glyth, glypt},
 {ulyph, guyph, gluph, glyuh, glypu},
 {vlyph, gvyph, glvph, glyvh, glypv},
 {wlyph, gwyph, glwph, glywh, glypw},
 {xlyph, gxyph, glxph, glyxh, glypx},
 {ylyph, gyyph, glyph, glyyh, glypy},
 {zlyph, gzyph, glzph, glyzh, glypz}}
```

Because of the extra nesting (`Table[Map[…]]`) we need to flatten the output at a deeper level; and delete duplicates.

```
Flatten[Map[Anagrams, blana, {2}]] // DeleteDuplicates
```

```
{phyla, haply, glyph, lymph, sylph}
```

Finally, put all the pieces together to create the function `Blanagrams`.

```
Blanagrams[word_String] := Module[{blana},
  blana = Table[
    Map[StringReplacePart[word, ch, {#, #}] &,
     Range[StringLength[word]]],
    {ch, CharacterRange["a", "z"]}];
  DeleteDuplicates[Flatten[Map[Anagrams, blana, {2}]]]
 ]
```

This turns out to be fairly quick for small words, but it bogs down for larger words.

```
Blanagrams["glyph"] // Timing
```

```
{1.185985, {phyla, haply, glyph, lymph, sylph}}
```

```
Blanagrams["zydeco"] // Timing
```

```
{8.548229, {zydeco, cloyed, comedy, decoys}}
```

We will wait until Section 12.3 to optimize this code by profiling (identifying slow computational chunks) and taking advantage of parallel processing built into *Mathematica*.

■ *Exercises*

1. Generalize the `RandomString` function to allow for a `Weights` option so that you can provide a weight for each character in the generated string. Include a rule to generate a message if the number of weights does not match the number of characters. For example:

```
RandomString[{"A", "T", "C", "G"}, 30,
 Weights → {.1, .2, .3, .4}]
```

```
GCGTCGTCGGGTCAGGTCCTCGTGTGGGCG
```

```
RandomString[{"A", "T", "C", "G"}, {5, 10},
 Weights → {.1, .4, .4, .1}]
```

```
{TTCACTTCCC, ACAACTGGCC, GATTCTTTTC, TGTCCTTTGA, TTCCTGCTGT}
```

```
RandomString[{"A", "T", "C", "G"}, {5, 10}, Weights → {.1, .4}]
```

```
⚠RandomString::badwt :
  The length of the list of weights must be the
    same as the length of the list of characters.
```

2. Write the function `Anagrams` developed in Section 9.2 without resorting to the use of `Permutations`. Consider using the `Sort` function to sort the characters. Note the difference in speed of the two approaches: one involving string functions and the other list functions that operate on lists of characters. Increase the efficiency of your search by only searching for words of the same length as your source word.

3. Rewrite the function `FindWordsContaining` using regular expressions instead of the patterns used in this section.

4. Using the text from several different sources, compute and then compare the number of punctuation characters per 1000 characters of text. `ExampleData["Text"]` gives a listing of many different texts that you can use.

5. The function `StringPartition` was developed specifically to deal with genomic data where one often needs uniformly-sized blocks to work with. Generalize `StringPartition` to fully accept the same argument structure as the built-in `Partition`.

6. Rewrite the text encoding example from Section 9.2 using `StringReplace` and regular expressions. First create an auxiliary function to encode a single character based on a key list of the form $\{\{pt_1, ct_1\}, \ldots\}$ where pt_i is a plaintext character and ct_i is its ciphertext encoding. For example, the pair $\{z, a\}$ would indicate the character z in the plaintext will be encoded as an a in the ciphertext. Then create an encoding function `encode[`*str*`, `*key*`]` using regular expressions to encode any string *str* using the *key* consisting of the plaintext/ciphertext character pairs.

■ References

[1] P. Wellin, *Programming with Mathematica®: An Introduction*, Cambridge, UK: Cambridge University Press, 2013. www.cambridge.org/wellin.

[2] A. Sinkov, *Elementary Cryptanalysis: A Mathematical Approach*, Washington, DC: The Mathematical Association of America, 1966.

[3] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners,* New York: Springer, 2010.

[4] J. Joyce, *Finnegans Wake*, New York: Viking Penguin Inc., 1939.

[5] Project Gutenberg. (Apr 3, 2013) www.gutenberg.org.

[6] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval,* New York: Cambridge University Press, 2008.

[7] Wolfram Research. “Working with String Patterns” from the Wolfram *Mathematica* Documentation Center—A Wolfram Web Resource.
reference.wolfram.com/mathematica/tutorial/WorkingWithStringPatternsOverview.html.

[8] Wolfram Research. “Regular Expression” from the Wolfram *Mathematica* Documentation Center—A Wolfram Web Resource.
reference.wolfram.com/mathematica/tutorial/RegularExpressions.html.

[9] University of Chicago Press, *The Chicago Manual of Style*, 16th ed., Chicago: University of Chicago Press, 2010.

[10] Centre for Applied Linguistics, University of Warwick. “British Academic Spoken English (BASE) and BASE Plus Collections.” (Apr 3, 2013) www2.warwick.ac.uk/fac/soc/al/research/collect/base.

[11] J. L. Borges, “The Library of Babel,” in *Labyrinths: Selected Stories & Other Writings,* New York: Modern Library, 1983.

[12] N. Cristianini and M. W. Hahn, *Introduction to Computational Genomics: A Case Studies Approach*. New York: Cambridge University Press, 2007.

[13] National Center for Biotechnology Information. “Nucleotide.” (Apr 3, 2013) www.ncbi.nlm.nih.gov/nuccore.

P. Wellin, “Strings,” *The Mathematica Journal*, 2013. dx.doi.org/doi:10.3888/tmj.15-4.

List of Additional Material

Additional electronic files:

1. www.mathematica-journal.com/data/uploads/2013/04/638154522.tar.gz
2. www.mathematica-journal.com/data/uploads/2013/04/StopWords.dat

About the Author

Paul Wellin worked for Wolfram Research from the early 1990s through 2011, directing the *Mathematica* training efforts with the Wolfram Education Group. Prior to that, he taught mathematics at both public school and at the university level for over 12 years. He has given talks, workshops, and seminars around the world on the integration of technical computing and education and he has served on numerous government advisory panels on these issues. He is the author or coauthor of several books on *Mathematica*. In addition to his writing, he is currently involved with several consulting projects, as well as book design and composition projects.

Paul Wellin
Chico, California
paulwellin@gmail.com