分子生物计算
(*Perl 语言编程*)

天津医科大学
生物医学工程与技术学院

2016-2017 学年上学期（秋）
2014 级生信班

第五章　基序和循环

伊现富（Yi Xianfu）

天津医科大学（TIJMU）
生物医学工程与技术学院

2016 年 11 月

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

Perl 语言基础

- 标量变量和数组变量
- 字符串操作（替换、翻译等）
- 从文件中读取数据

DNA 和蛋白质生物序列数据的处理

- 拼接 DNA 片段
- 把 DNA 转录成 RNA
- 获取反向互补序列
- 从文件中读取序列

Perl 语言基础

- 标量变量和数组变量
- 字符串操作（替换、翻译等）
- 从文件中读取数据

DNA 和蛋白质生物序列数据的处理

- 拼接 DNA 片段
- 把 DNA 转录成 RNA
- 获取反向互补序列
- 从文件中读取序列

Perl 语言基础

- 根据条件测试的结果采取不同的行动
- 使用循环
- 通过键盘与用户进行交互
- 使用基本的正则表达式
- 操作字符串和数组
- 把处理结果写入文件

DNA 和蛋白质序列数据的处理

- 使用循环读取文件中的序列数据
- 查找蛋白质序列中的基序
- 计算核苷酸频率

Perl 语言基础

- 根据条件测试的结果采取不同的行动
- 使用循环
- 通过键盘与用户进行交互
- 使用基本的正则表达式
- 操作字符串和数组
- 把处理结果写入文件

DNA 和蛋白质序列数据的处理

- 使用循环读取文件中的序列数据
- 查找蛋白质序列中的基序
- 计算核苷酸频率

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

定义

控制流程（也称为流程控制，flow control）是电脑运算领域的用语，意指在程序运行时，个别的指令（或是陈述、子程序）运行或求值的顺序。

指令

不同的编程语言所提供的流程控制指令也会随之不同，但一般可以分为以下几种：

- 继续运行位于不同位置的一段指令（无条件分支指令）。
- 若特定条件成立时，运行一段指令，例如 C 语言的 switch 指令，是一种有条件分支指令。
- 运行一段指令若干次，直到特定条件成立为止，例如 C 语言的 for 指令，仍然可视为一种有条件分支指令。
- 运行位于不同位置的一段指令，但完成后会继续运行原来要运行的指令，包括子程序、协程（coroutine）及延续性（continuation）。
- 停止程序，不运行任何指令（无条件的终止）。

定义

控制流程（也称为流程控制，flow control）是电脑运算领域的用语，意指在程序运行时，个别的指令（或是陈述、子程序）运行或求值的顺序。

指令

不同的编程语言所提供的流程控制指令也会随之不同，但一般可以分为以下几种：

- 继续运行位于不同位置的一段指令（无条件分支指令）。
- 若特定条件成立时，运行一段指令，例如 C 语言的 switch 指令，是一种有条件分支指令。
- 运行一段指令若干次，直到特定条件成立为止，例如 C 语言的 for 指令，仍然可视为一种有条件分支指令。
- 运行位于不同位置的一段指令，但完成后会继续运行原来要运行的指令，包括子程序、协程（coroutine）及延续性（continuation）。
- 停止程序，不运行任何指令（无条件的终止）。

默认

除非明确指明不按顺序执行，否则程序将从最顶端的第一个语句开始，顺序执行到最底端的最后一个语句。

分类

- 条件判断：只在条件测试成功的前提下执行相应的语句，否则直接跳过这些语句
- 循环：一直重复语句，直到相应的测试失败为止

默认

除非明确指明不按顺序执行，否则程序将从最顶端的第一个语句开始，顺序执行到最底端的最后一个语句。

分类

- 条件判断：只在条件测试成功的前提下执行相应的语句，否则直接跳过这些语句
- 循环：一直重复语句，直到相应的测试失败为止

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

Perl 中的实现

- if
- if-else
- unless

工作原理

- 条件测试的结果为真（true）或者为假（false）
- 如果测试结果为真，后面的语句就会被执行
- 如果测试结果为假，后面的语句就不会执行，直接被跳过

Perl 中的实现

- if
- if-else
- unless

工作原理

- 条件测试的结果为真（true）或者为假（false）
- 如果测试结果为真，后面的语句就会被执行
- 如果测试结果为假，后面的语句就不会执行，直接被跳过

```
1 # =表示赋值, ==表示数字相等
2 if( 1 == 1 ) {
3   print "1 equals 1\n\n";
4 }
5
6 if( 1 ) {
7   print "1 evaluates to true\n\n";
8 }
```

```
1 if( 1 == 0 ) {
2   print "1 equals 0\n\n";
3 }
4
5 if( 0 ) {
6   print "0 evaluates to true\n\n";
7 }
```

```
1  # 两种写法完全等价
2  # 比较“标准”，易读易懂
3  if( 1 == 1 ) {
4     print "1 equals 1\n\n";
5  }
6
7  # 更加简洁
8  print "1 equals 1\n\n" if (1 == 1);
9
10 # 类比自然语言（英语）
11 # If you build it, they will come.
12 # They will come if you build it.
```

```
1 if( 1 == 1 ) {
2   print "1 equals 1\n\n";
3 } else {
4   print "1 does not equal 1\n\n";
5 }
6
7 # 1 equals 1
```

```
1 if( 1 == 0 ) {
2   print "1 equals 0\n\n";
3 } else {
4   print "1 does not equal 0\n\n";
5 }
6
7 # 1 does not equal 0
```

unless

- unless 和 if 完全相反
- 如果测试结果为真，语句会直接被跳过而不执行
- 如果测试结果为假，相应的语句会被执行

```
1 unless( 1 == 0 ) {
2   print "1 does not equal 0\n\n";
3 }
4 # 1 does not equal 0
5
6 # 类比自然语言
7 # Unless you study Russian literature, you
    are ignorant of Chekov.
8 # 除非下课铃响了，否则我们不会下课。
```

unless

- unless 和 if 完全相反
- 如果测试结果为真，语句会直接被跳过而不执行
- 如果测试结果为假，相应的语句会被执行

```
1 unless( 1 == 0 ) {
2   print "1 does not equal 0\n\n";
3 }
4 # 1 does not equal 0
5
6 # 类比自然语言
7 # Unless you study Russian literature, you
    are ignorant of Chekov.
8 # 除非下课铃响了，否则我们不会下课。
```

```
1  unless ( 1 == 0 ) {
2    print "1 does not equal 0\n\n";
3  }
4  # 1 does not equal 0
5
6  if ( 1 != 0 ) {
7    print "1 does not equal 0\n\n";
8  }
9  # 1 does not equal 0
10
11 if ( !(1 == 0) ) {
12   print "1 does not equal 0\n\n";
13 }
14 # 1 does not equal 0
```

条件测试

- 数字比较：==, !=, <, >, <=, >=
- 字符串比较：eq, ne, lt, gt, le, ge
- 文件测试：-e, -s, -z, -f, -d, -l, -r, -w, -x, ...
- 变量测试：真（True），假（False）

说谎者悖论

这个语句为假。

这个语句不为真。

这个语句只为假。

下个语句为真。
上个语句为假。

第二个语句为假。
第三个语句为假。
第一个语句为假。

真 vs. 假

- 真假判定
  - 如果是数字：0（0, 0.0, -0.0, ……）为假，其他所有数字都为真
  - 如果是字符串：空字符串（用 "" 或 '' 表示）为假，其他所有字符串都为真【参看下一条目】
  - 注意：字符串 '0' 或 "0"（和数字 0 是同一个标量值）是唯一被当成假的非空字符串
  - 如果既不是数字也不是字符串，先转换成数字或字符串再行判断
- 补充说明
  - "", ''（空字符串）vs. " ", ' '（空白/格字符串）
  - 没有借阅证 vs. 有借阅证但没有借阅记录

括号成对

- （小、中、大、尖）括号成对是最普遍的编程特性
- 左右括号的数目相等且都出现在正确的位置
- 括号不配对或者没有在正确的位置是非常常见的语法错误
- 保证括号配对的方法：
  - 每行/块代码做的事情不要太多
  - 使用缩进使代码块明显一些
  - 使用专用的文本编辑器（Vim 中的%）
  - 对代码进行格式化（perltidy）

```perl
#!/usr/bin/perl -w

$word = 'MNIDDKL';

if($word eq 'QSTVSGE') {
  print "QSTVSGE\n";
} elsif($word eq 'MRQQDMISHDEL') {
  print "MRQQDMISHDEL\n";
} elsif ( $word eq 'MNIDDKL'  ) {
  print "MNIDDKL--the magic word!\n";
} else {
  print "Is \"$word\" a peptide? This program
   is not sure.\n";
}

exit;
```

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

Perl 中的实现

- while
- for
- foreach
- until

工作原理

只要测试为真，就会重复执行被成对大括号包裹起来的语句块。

Perl 中的实现

- while
- for
- foreach
- until

工作原理

只要测试为真，就会重复执行被成对大括号包裹起来的语句块。

while vs. until

- until 只不过是一个改装过的 while 循环罢了。
- 唯一差别：until 会在条件为假时重复执行，而不是为真时执行。
- 类似 if 和 unless 转化的例子，你可以用否定条件表达式的方法，把任意一个 until 循环改写成 while 循环。

for vs. foreach

- 在 Perl 解析器里，foreach 和 for 这两个关键字实际上等价的。当 Perl 看到其中之一时，就好像看到了另一个。
- Perl 可以从圆括号里的内容判断出你的意图：如果里面有两个分号，它就是 for 循环；若没有分号，就说明它是一个 foreach 循环。
- 在 Perl 世界里，纯正的 foreach 循环几乎总是更好地选择。

while vs. until

- until 只不过是一个改装过的 while 循环罢了。
- 唯一差别：until 会在条件为假时重复执行，而不是为真时执行。
- 类似 if 和 unless 转化的例子，你可以用否定条件表达式的方法，把任意一个 until 循环改写成 while 循环。

for vs. foreach

- 在 Perl 解析器里，foreach 和 for 这两个关键字实际上等价的。当 Perl 看到其中之一时，就好像看到了另一个。
- Perl 可以从圆括号里的内容判断出你的意图：如果里面有两个分号，它就是 for 循环；若没有分号，就说明它是一个 foreach 循环。
- 在 Perl 世界里，纯正的 foreach 循环几乎总是更好地选择。

```
1  $count = 0;
2  until ( $count >= 10 ) {
3    print "count is now $count\n";
4    $count += 2;
5  }
6
7  $count = 0;
8  while ( $count < 10 ) {
9    print "count is now $count\n";
10   $count += 2;
11 }
12
13 $count = 0;
14 while ( !($count >= 10) ) {
15   print "count is now $count\n";
16   $count += 2;
17 }
```

```
1  for ( 1..10 ) {
2    print "I can count to $_!\n";
3  }
4
5  foreach ( 1..10 ) {
6    print "I can count to $_!\n";
7  }
8
9  for ( $i=1; $i<=10; $i++ ) {
10   print "I can count to $i!\n";
11 }
```

```
1  #!/usr/bin/perl -w
2  # Example 5-2   Reading protein sequence data from
       a file, take 4
3
4  # The filename of the file containing the protein
     sequence data
5  $proteinfilename = 'NM_021964fragment.pep';
6
7  # First we have to "open" the file, and in case
     the
8  # open fails, print an error message and exit the
     program.
9  unless ( open( PROTEINFILE, $proteinfilename ) ) {
10
11     print "Could not open file $proteinfilename!\n
     ";
12     exit;
13 }
```

```
15 # Read the protein sequence data from the
     file in a "while" loop,
16 # printing each line as it is read.
17 while ( $protein = <PROTEINFILE> ) {
18
19     print "  ######  Here is the next line of
     the file:\n";
20
21     print $protein;
22 }
23
24 # Close the file.
25 close PROTEINFILE;
26
27 exit;
```

```
1   ######  Here is the next line of the file:
2 MNIDDKLEGLFLKCGGIDEMQSSRTMVVMGGVSGQSTVSGELQD
3   ######  Here is the next line of the file:
4 SVLQDRSMPHQEILAADEVLQESEMRQQDMISHDELMVHEETVKNDEEQMETHERLPQ
5   ######  Here is the next line of the file:
6 LQYALNVPISVKQEITFTDVSEQLMRDKKQIR
```

```
1  #!/usr/bin/perl -w
2
3  $proteinfilename = 'NM_021964fragment.pep';
4
5  unless ( open( PROTEINFILE, $proteinfilename ) ) {
6      print "Could not open file $proteinfilename!\n";
7      exit;
8  }
9
10 while ( $protein = <PROTEINFILE> ) {
11     print "  ######  Here is the next line of the file:\
       n";
12     print $protein;
13 }
14
15 close PROTEINFILE;
16
17 exit;
```

open

- open：打开文件，是一个系统调用
- 一定要检查系统调用（此处是打开文件）的成功与否
- 打开文件失败时，要立即告知用户，退出程序

unless

- unless 与 if 相反
- 如果成功打开文件，open 系统调用会返回真
- 如果 open 系统调用失败，代码块就会执行：输出错误信息、退出程序

open

- open：打开文件，是一个系统调用
- 一定要检查系统调用（此处是打开文件）的成功与否
- 打开文件失败时，要立即告知用户，退出程序

unless

- unless 与 if 相反
- 如果成功打开文件，open 系统调用会返回真
- 如果 open 系统调用失败，代码块就会执行：输出错误信息、退出程序

- 条件和循环是编程语言最强大的特性之一
- 条件可以使程序适应不同的状况，针对不同的输入采取不同的方案（人工智能）
- 利用循环，仅仅使用几行代码，就可以处理大量的输入，对计算进行重复迭代与提炼

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

```
1 while ( $alive ) {
2   if ( $needs_nutrients ) {
3     print "Cell needs nutrients\n";
4   }
5 }
```

```
1 while ( $alive )
2 {
3   if ( $needs_nutrients )
4   {
5     print "Cell needs nutrients\n";
6   }
7 }
```

```
1 while ( $alive ) {
2   if ( $needs_nutrients ) {
3     print "Cell needs nutrients\n";
4   }
5 }
```

```
1 while ( $alive )
2 {
3   if ( $needs_nutrients )
4   {
5     print "Cell needs nutrients\n";
6   }
7 }
```

```
1    while ( $alive )
2      {
3       if ( $needs_nutrients )
4 {
5   print "Cell needs nutrients\n";
6 }
7 }
```

```
1 while($alive){if($needs_nutrients){print "
  Cell needs nutrients\n"; }}
```

```
1    while ( $alive )
2     {
3      if ( $needs_nutrients )
4 {
5   print "Cell needs nutrients\n";
6 }
7 }
```

```
1 while($alive){if($needs_nutrients){print "
  Cell needs nutrients\n"; }}
```

- Perl 只关心句法元素的正确顺序，不依赖语句布局
- 推荐格式 A 和 B，不推荐格式 C 和 D
- 推荐使用 perltidy 对代码进行格式化
- Perl 的风格指南：`perldoc perlstyle`

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

sequence motif: a sequence pattern of nucleotides in a DNA sequence or amino acids in a protein

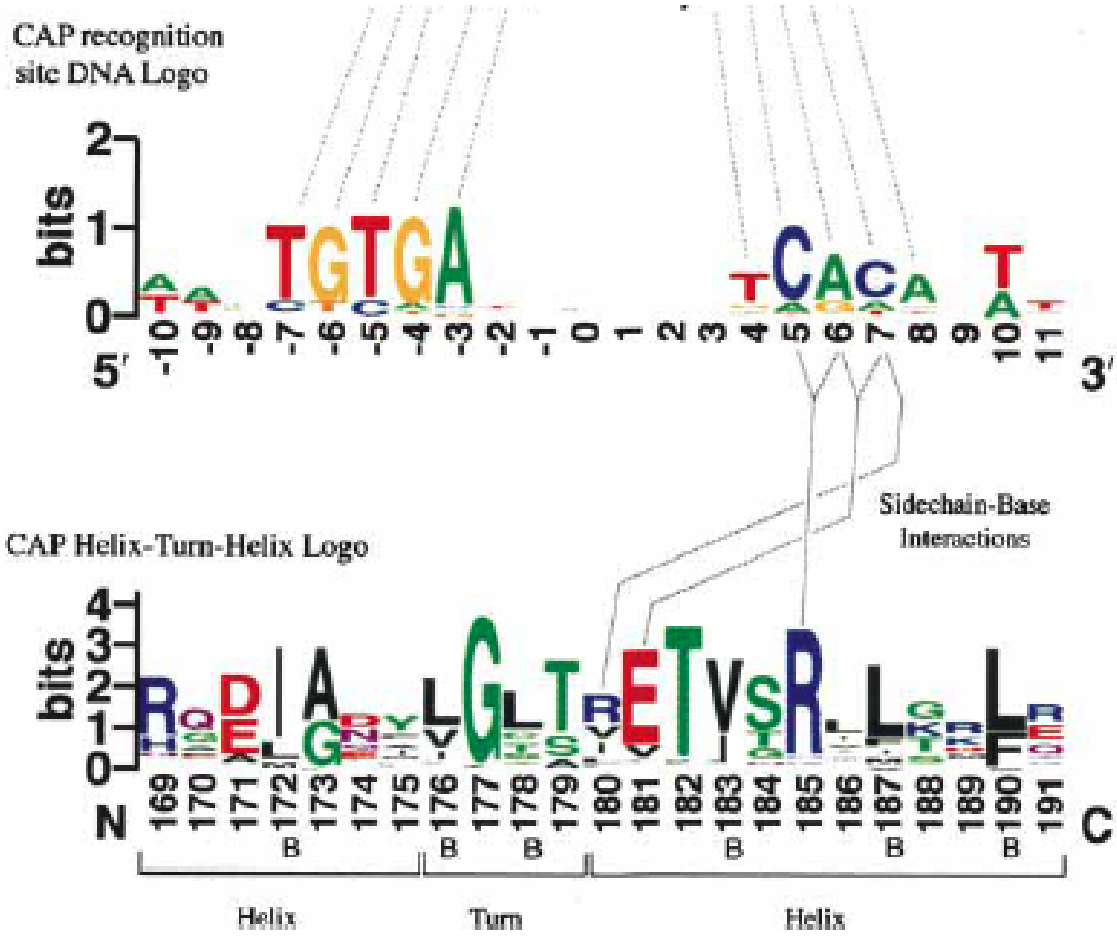
- In genetics, a sequence motif is a nucleotide or amino-acid sequence pattern that is widespread and has, or is conjectured to have, a biological significance.
- For proteins, a sequence motif is distinguished from a structural motif, a motif formed by the three-dimensional arrangement of amino acids which may not be adjacent.

structural motif: a pattern in a protein structure formed by the spatial arrangement of amino acids

- In a chain-like biological molecule, such as a protein or nucleic acid, a structural motif is a supersecondary structure, which also appears in a variety of other molecules.
- Motifs do not allow us to predict the biological functions: they are found in proteins and enzymes with dissimilar functions.

基序（motif）

特定的 DNA 片段（如 DNA 调控元件）或蛋白质片段（如保守区域）



基序特点

- 往往不是一个特定的序列
- 可能有变体（某个位置上的碱基或者残基是什么并不重要）
- 可能有不同的长度

解决方案

- 用正则表达式表征基序
- 在字符串中进行查找

基序特点

- 往往不是一个特定的序列
- 可能有变体（某个位置上的碱基或者残基是什么并不重要）
- 可能有不同的长度

解决方案

- 用正则表达式表征基序
- 在字符串中进行查找

```
1  #!/usr/bin/perl -w
2  # Example 5-3   Searching for motifs
3
4  # Ask the user for the filename of the file
     containing
5  # the protein sequence data, and collect it
     from the keyboard
6  print "Please type the filename of the
     protein sequence data: ";
7
8  $proteinfilename = <STDIN>;
9
10 # Remove the newline from the protein
     filename
11 chomp $proteinfilename;
```

```
13 # open the file, or exit
14 unless ( open( PROTEINFILE, $proteinfilename ) ) {
15
16     #print "Cannot open file '$proteinfilename'\n\n
     ";
17     print "Cannot open file \"$proteinfilename\"\n
     \n";
18     exit;
19 }
20
21 # Read the protein sequence data from the file,
     and store it
22 # into the array variable @protein
23 @protein = <PROTEINFILE>;
24
25 # Close the file - we've read all the data into
     @protein now.
26 close PROTEINFILE;
```

```
27 # Put the protein sequence data into a single
       string, as it's easier
28 # to search for a motif in a string than in
     an array of
29 # lines (what if the motif occurs over a line
       break?)
30 $protein = join( '', @protein );
31
32 # Remove whitespace
33 $protein =~ s/\s//g;
34
35 # In a loop, ask the user for a motif, search
       for the motif,
36 # and report if it was found.
37 # Exit if no motif is entered.
```

```
38 do {
39     print "Enter a motif to search for: ";
40
41     $motif = <STDIN>;
42
43     # Remove the newline at the end of $motif
44
45     chomp $motif;
46
47     # Look for the motif
```

```
49      if ( $protein =~ /$motif/ ) {
50     #if ( $protein =~ m/$motif/ ) {
51          print "I found it!\n\n";
52      }
53      else {
54          print "I couldn\'t find it.\n\n";
55         #print "I couldn't find it.\n\n";
56      }
57
58      # exit on an empty user input
59  } until ( $motif =~ /^\s*$/ );
60
61  # exit the program
62  exit;
```

```
1  Please type the filename of the protein sequence
     data:
2  NM_021964fragment.pep
3  Enter a motif to search for: SVLQ
4  I found it!
5
6  Enter a motif to search for: jkl
7  I couldn't find it.
8
9  Enter a motif to search for: QDSV
10 I found it!
11
12 Enter a motif to search for: HERLPQGLQ
13 I found it!
14
15 Enter a motif to search for:
16 I couldn't find it.
```

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

```
1 # 文件句柄可以和文件相关联
2 @protein = <PROTEINFILE>;
3
4 # 文件句柄也可以和键盘输入相关联
5 $proteinfilename = <STDIN>;
6
7 # 去掉字符串末尾的换行符
8 chomp $proteinfilename;
```

chomp vs. chop

- chomp 会去掉字符串末尾的换行符（有则去，没有则不进行任何处理）
- chop 删除字符串末尾的最后一个字符（不管最后一个字符是什么，都会被去掉）

```
1 # 文件句柄可以和文件相关联
2 @protein = <PROTEINFILE>;
3
4 # 文件句柄也可以和键盘输入相关联
5 $proteinfilename = <STDIN>;
6
7 # 去掉字符串末尾的换行符
8 chomp $proteinfilename;
```

chomp vs. chop

- chomp 会去掉字符串末尾的换行符（有则去，没有则不进行任何处理）
- chop 删除字符串末尾的最后一个字符（不管最后一个字符是什么，都会被去掉）

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

```perl
$protein = join ( '', @protein );
```

join

- 把数组中的元素合并成一个字符串，元素之间用指定的字符串进行分隔
- 此处用于分隔元素的字符串是空字符串：''（使用 "" 亦可）
- join 可以处理数组或者标量列表

```perl
# 连接两个DNA片段的一种方法
$DNA3 = $DNA1 . $DNA2;
# 又又又一种方法
$DNA3 = join ( "", ($DNA1, $DNA2) );
```

```perl
$protein = join ( '', @protein );
```

join

- 把数组中的元素合并成一个字符串，元素之间用指定的字符串进行分隔
- 此处用于分隔元素的字符串是空字符串：''（使用 "" 亦可）
- join 可以处理数组或者标量列表

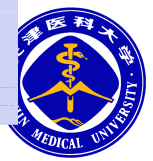
```perl
# 连接两个DNA片段的一种方法
$DNA3 = $DNA1 . $DNA2;
# 又又又一种方法
$DNA3 = join ( "", ($DNA1, $DNA2) );
```

```perl
$protein = join ( '', @protein );
```

join

- 把数组中的元素合并成一个字符串，元素之间用指定的字符串进行分隔
- 此处用于分隔元素的字符串是空字符串：''（使用 "" 亦可）
- join 可以处理数组或者标量列表

```perl
# 连接两个DNA片段的一种方法
$DNA3 = $DNA1 . $DNA2;
# 又又又一种方法
$DNA3 = join ( "", ($DNA1, $DNA2) );
```



教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

do-until

先执行一次代码块，之后再进行条件测试。（不是先测试后执行）

工作流程

1. 提示用户输入要查找的基序
2. 获取用户的输入
3. 查找基序并报告查找结果
4. 重复上述步骤之前，测试用户是否输入了一个空行
5. 如果输入的是空行，退出循环

do-until

先执行一次代码块，之后再进行条件测试。（不是先测试后执行）

工作流程

1. 提示用户输入要查找的基序
2. 获取用户的输入
3. 查找基序并报告查找结果
4. 重复上述步骤之前，测试用户是否输入了一个空行
5. 如果输入的是空行，退出循环

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

- 可以轻松处理各种各样的字符串（包括 DNA 和蛋白质序列数据）
- 使用元字符来匹配一个或多个字符串
- 可以非常简单（比如匹配单词本身的一个单词，`/Perl/`, `/cell/`, `/computer/`, `/bioinformatics/`）
- 也可以非常复杂（比如匹配一个大的单词集合，甚至每一个单词）

```
1 $protein =~ s/\s//g;
```

说明

- 作用：删除序列中的换行符等非序列数据的字符（此处是非打印字符）
- `s/\s//g`：把空白字符全部替换成空字符串（其实就是删除所有的空白字符）
- 元字符 `\s`：匹配空格、制表符、换行符、换页符和回车符
- 字符集 `[ \t\n\f\r]`：分别表示空格、制表符、换行符、换页符和回车符
- 元字符 `\s` 等同于字符集 `[ \t\n\f\r]`
- `s///`中的前两个斜线之间：放置正则表达式（C、`\s` 等）

```
1 $protein =~ s/\s//g;
```

说明

- 作用：删除序列中的换行符等非序列数据的字符（此处是非打印字符）
- `s/\s//g`：把空白字符全部替换成空字符串（其实就是删除所有的空白字符）
- 元字符 `\s`：匹配空格、制表符、换行符、换页符和回车符
- 字符集 `[ \t\n\f\r]`：分别表示空格、制表符、换行符、换页符和回车符
- 元字符 `\s` 等同于字符集 `[ \t\n\f\r]`
- `s///`中的前两个斜线之间：放置正则表达式（C、`\s` 等）

```
1 } until ( $motif =~ /^\s*$/ );
```

说明

- 作用：检测 `$motif` 变量中的空行
- 解释：从开头（`^`）到结尾（`$`）只有零个或者多个（`*`）空白字符（`\s`）的字符串

```
1 /A[DS]V/
2 /KND*E{2,}/
3 /EE.*EE/
```

```
1 } until ( $motif =~ /^\s*$/ );
```

说明

- 作用：检测 `$motif` 变量中的空行
- 解释：从开头（`^`）到结尾（`$`）只有零个或者多个（`*`）空白字符（`\s`）的字符串

```
1 /A[DS]V/
2 /KND*E{2,}/
3 /EE.*EE/
```

```
1 } until ( $motif =~ /^\s*$/ );
```

说明

- 作用：检测 `$motif` 变量中的空行
- 解释：从开头（`^`）到结尾（`$`）只有零个或者多个（`*`）空白字符（`\s`）的字符串

```
1 /A[DS]V/
2 /KND*E{2,}/
3 /EE.*EE/
```

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

```
1 if ( $protein =~ /$motif/ ) {
2 #if ( $protein =~ m/$motif/ ) {
```

说明

- 绑定操作符 =~ 指定在 $protein 中进行查找
- /$motif/指定查找 $motif 变量中的正则表达式（此处是基序）
- 变量内插：把变量的值插入到字符串中（就像你直接把该字符串放在此处一样）
- 使用变量内插比直接放置字符串更加灵活

```
1 if ( $protein =~ /$motif/ ) {
2 #if ( $protein =~ m/$motif/ ) {
```

说明

- 绑定操作符 =~ 指定在 $protein 中进行查找
- /$motif/指定查找 $motif 变量中的正则表达式（此处是基序）
- 变量内插：把变量的值插入到字符串中（就像你直接把该字符串放在此处一样）
- 使用变量内插比直接放置字符串更加灵活

```
#!/usr/bin/perl -w
$proteinfilename = <STDIN>;
chomp $proteinfilename;
unless ( open( PROTEINFILE, $proteinfilename ) ) {
    print "Cannot open file \"$proteinfilename\"\n\n"; exit;
}
@protein = <PROTEINFILE>;
close PROTEINFILE;
$protein = join( '', @protein );
$protein =~ s/\s//g;
do {
    print "Enter a motif to search for: ";
    $motif = <STDIN>;
    chomp $motif;
    if ( $protein =~ /$motif/ ) {
        print "I found it!\n\n";
    }
    else {
        print "I couldn\'t find it.\n\n";
    }
} until ( $motif =~ /^\s*$/ );
exit;
```

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

```
5'- GCT TAC CGC CCC AGT GAG ACC CTG TGC
GGC GGG GAG CTG GTG GAC ACC CTC CAG TTC
GTC TGT GGG GAC CGC GGC TTC TAC TTC AGC
AGG CCC GCA AGC CGT GTG AGC CGT CGC AGC
CGT GGC ATC GTT GAG GAG TGC TGT TTC CGC
AGC TGT GAC CTG GCC CTC CTG GAG ACG TAC
TGT GCT ACC CCC GCC AAG TCC GAG -3'.
```

问题

- 编码还是不编码？
- 是否含有调控元件？
- 与其他已知的 DNA 序列是否相关？
- 四种核苷酸的数目和比例是多少？
- GC 含量如何？
- ……

```
5'- GCT TAC CGC CCC AGT GAG ACC CTG TGC
GGC GGG GAG CTG GTG GAC ACC CTC CAG TTC
GTC TGT GGG GAC CGC GGC TTC TAC TTC AGC
AGG CCC GCA AGC CGT GTG AGC CGT CGC AGC
CGT GGC ATC GTT GAG GAG TGC TGT TTC CGC
AGC TGT GAC CTG GCC CTC CTG GAG ACG TAC
TGT GCT ACC CCC GCC AAG TCC GAG -3'.
```

问题

- 编码还是不编码？
- 是否含有调控元件？
- 与其他已知的 DNA 序列是否相关？
- 四种核苷酸的数目和比例是多少？
- GC 含量如何？
- ……

```
1  for each base in the DNA
2    if base is A
3      count_of_A = count_of_A + 1
4    if base is C
5      count_of_C = count_of_C + 1
6    if base is G
7      count_of_G = count_of_G + 1
8    if base is T
9      count_of_T = count_of_T + 1
10 done
11
12 print count_of_A, count_of_C, count_of_G,
     count_of_T
```

通用策略——策略一

把 DNA 拆解成单个碱基，存储到数组中，对数组中的元素进行迭代处理

```
1 数组：A C G T G T A C
2 索引：0 1 2 3 4 5 6 7
```

通用策略——策略二

对 DNA 字符串中的位置（索引）进行迭代处理

```
1 碱基：ACGTGTAC
2 位置：12345678
```

专有策略

- 策略三，策略四，……

通用策略——策略一

把 DNA 拆解成单个碱基，存储到数组中，对数组中的元素进行迭代处理

```
1 数组：A C G T G T A C
2 索引：0 1 2 3 4 5 6 7
```

通用策略——策略二

对 DNA 字符串中的位置（索引）进行迭代处理

```
1 碱基：ACGTGTAC
2 位置：12345678
```

专有策略

- 策略三，策略四，……

通用策略——策略一

把 DNA 拆解成单个碱基，存储到数组中，对数组中的元素进行迭代处理

```
1 数组：A C G T G T A C
2 索引：0 1 2 3 4 5 6 7
```

通用策略——策略二

对 DNA 字符串中的位置（索引）进行迭代处理

```
1 碱基：ACGTGTAC
2 位置：12345678
```

专有策略

- 策略三，策略四，……

```
1 # small.dna
2 AAAAAAAAAAAAAAGGGGGGGTTTTCCCCCCCC
3 CCCCCGTCGTAGTAAAGTATGCAGTAGCVG
4 CCCCCCCCCCGGGGGGGGAAAAAAAAAAAAAATTTTTTAT
5 AAACG
```

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

- 把字符串拆解成数组：把字符串中的每一个字符分离开来（相当于把一句话分成单个的字）
- 拆解与 join 相反：join 把数组中的字符串合并成单个的标量值
- 把 DNA 字符串拆解：把 DNA 序列中的每个碱基都分离出来
- 拆解成数组：按照顺序，每个碱基都成为了数组中的元素

```
1  read in the DNA from a file
2
3  join the lines of the file into a single
     string $DNA
4
5  # make an array out of the bases of $DNA
6  @DNA = explode $DNA
7
8  # initialize the counts
9  count_of_A = 0
10 count_of_C = 0
11 count_of_G = 0
12 count_of_T = 0
```

```
1  for each base in @DNA
2
3    if base is A
4      count_of_A = count_of_A + 1
5    if base is C
6      count_of_C = count_of_C + 1
7    if base is G
8      count_of_G = count_of_G + 1
9    if base is T
10     count_of_T = count_of_T + 1
11 done
12
13 print count_of_A, count_of_C, count_of_G,
     count_of_T
```

```
1  #!/usr/bin/perl -w
2  # Example 5-4   Determining frequency of
     nucleotides
3
4  # Get the name of the file with the DNA
     sequence data
5  print "Please type the filename of the DNA
     sequence data: ";
6
7  $dna_filename = <STDIN>;
8
9  # Remove the newline from the DNA filename
10 chomp $dna_filename;
```

```
12 # open the file, or exit
13 unless ( open( DNAFILE, $dna_filename ) ) {
14
15     print "Cannot open file \"$dna_filename
     \"\n\n";
16     exit;
17 }
18
19 # Read the DNA sequence data from the file,
     and store it
20 # into the array variable @DNA
21 @DNA = <DNAFILE>;
22
23 # Close the file
24 close DNAFILE;
```

```
26 # From the lines of the DNA file,
27 # put the DNA sequence data into a single string.
28 $DNA = join( '', @DNA );
29
30 # Remove whitespace
31 $DNA =~ s/\s//g;
32
33 # Now explode the DNA into an array where each
     letter of the
34 # original string is now an element in the array.
35 # This will make it easy to look at each position.
36 # Notice that we're reusing the variable @DNA for
     this purpose.
37 @DNA = split( '', $DNA );
```

```
39 # Initialize the counts.
40 # Notice that we can use scalar variables to
     hold numbers.
41 $count_of_A = 0;
42 $count_of_C = 0;
43 $count_of_G = 0;
44 $count_of_T = 0;
45 $errors     = 0;
46
47 # In a loop, look at each base in turn,
     determine which of the
48 # four types of nucleotides it is, and
     increment the
49 # appropriate count.
```

```
50 foreach $base (@DNA) {
51 #for $base (@DNA) {
52     if ( $base eq 'A' ) {
53         ++$count_of_A;
54     }
55     elsif ( $base eq 'C' ) {
56         ++$count_of_C;
57     }
58     elsif ( $base eq 'G' ) {
59         ++$count_of_G;
60     }
61     elsif ( $base eq 'T' ) {
62         ++$count_of_T;
63     }
64     else {
65         print "!!!!!!!! Error - I don\'t recognize this
   base: $base\n";
66         ++$errors;
67     }
68 }
```

```
70 # print the results
71 print "A = $count_of_A\n";
72 print "C = $count_of_C\n";
73 print "G = $count_of_G\n";
74 print "T = $count_of_T\n";
75 print "errors = $errors\n";
76
77 # exit the program
78 exit;
```

```
1 Please type the filename of the DNA sequence
    data: small.dna
2 !!!!!!!! Error - I don't recognize this base:
     V
3
4 A = 40
5 C = 27
6 G = 24
7 T = 17
```

```
1 @DNA = split( '', $DNA );
```

split

- split 与 join 是相对的，一个拆解、一个合并
- split 会以指定的字符串（第一个参数）为分隔符来拆解字符串（第二个参数）
- 如果第一个参数是空字符串，split 会把字符串拆解成单个的字符
- 此处是把 DNA 序列拆解成了单个碱基

```
1 @DNA = split( '', $DNA );
```

split

- split 与 join 是相对的，一个拆解、一个合并
- split 会以指定的字符串（第一个参数）为分隔符来拆解字符串（第二个参数）
- 如果第一个参数是空字符串，split 会把字符串拆解成单个的字符
- 此处是把 DNA 序列拆解成了单个碱基

- 初始化（initialization）一个变量的值意味着在声明该变量后给它一个值
- 如果不初始化变量，它的值将被假定为 `'undef'`
- 对于 `'undef'` 的变量：
  - 如果在数字上下文中使用，它的值为 0
  - 如果在字符串上下文中使用，它的值为空字符串
- Perl 程序员通常不去初始化变量
- 养成初始化变量的习惯，使程序更加易读、易维护

- 声明（declare）变量：指定变量的名字与属性
- 变量的属性包括：初始值、作用域和变量类型（在 Perl 中不需要）等
- 在 Perl 中，在使用变量之前对其进行声明并不是必需的
- Perl 在判断标量变量是什么时非常智能（打印输出 vs. 算术运算；字符串 vs. 数字）

```
1  #!/usr/bin/perl -w
2  # Example 5-5   Demonstration of Perl's built
     -in knowledge about numbers and strings
3
4  $num = 1234;
5
6  $str = '1234';
7
8  # print the variables
9  print $num, " ", $str, "\n";
10 #print "$num $str\n";
```

```
11 # add the variables as numbers
12 $num_or_str = $num + $str;
13
14 print $num_or_str, "\n";
15
16 # concatenate the variables as strings
17 $num_or_str = $num . $str;
18
19 print $num_or_str, "\n";
20
21 exit;
```

```
1 1234 1234
2 2468
3 12341234
```

智能化处理

- Perl 能对标量变量的数据类型进行智能化处理
- 当你需要数字时（比如算数运算），Perl 就会把它当做数字来处理
- 当你需要字符串时（比如拼接），Perl 就会把它当做字符串来处理

```
1 1234 1234
2 2468
3 12341234
```

智能化处理

- Perl 能对标量变量的数据类型进行智能化处理
- 当你需要数字时（比如算数运算），Perl 就会把它当做数字来处理
- 当你需要字符串时（比如拼接），Perl 就会把它当做字符串来处理

```perl
#!/usr/bin/perl -w

$num = 1234; $mix = '12ab';
# print the variables
print "$num $mix\n"; #1234 12ab

# add the variables as numbers
$sum = $num + $mix;
print $sum, "\n"; #1246
#Argument "12ab" isn't numeric in addition
  (+) at xxx.pl line NNN.

# concatenate the variables as strings
$cat = $num . $mix;
print $cat, "\n"; #123412ab
```

```perl
#!/usr/bin/perl -w

$num = 1234;
$str = '1234';
print $num, " ", $str, "\n";

# add the variables as numbers
$num_or_str = $num + $str;
print $num_or_str, "\n";

# concatenate the variables as strings
$num_or_str = $num . $str;
print $num_or_str, "\n";

exit;
```

```
1 foreach $base (@DNA) {
2 #for $base (@DNA) {
```

foreach

- 使用 foreach 对 @DNA 数组中的元素进行循环处理
- 每循环一次，$base 就会被设定成数组的下一个元素
- 如果不指明标量变量（此处是 $base），Perl 会使用特殊变量 $_
- 在不提供参数的情况下，Perl 的许多内置函数都会对特殊变量进行操作（此处的 foreach，模式匹配，print;，……）

```
1 foreach $base (@DNA) {
2 #for $base (@DNA) {
```

foreach

- 使用 foreach 对 `@DNA` 数组中的元素进行循环处理
- 每循环一次，`$base` 就会被设定成数组的下一个元素
- 如果不指明标量变量（此处是 `$base`），Perl 会使用特殊变量 `$_`
- 在不提供参数的情况下，Perl 的许多内置函数都会对特殊变量进行操作（此处的 foreach，模式匹配，`print;`，……）

```
1  foreach (@DNA) {
2    if ( /A/ ) {
3      ++$count_of_A;
4    } elsif ( /C/  ) {
5      ++$count_of_C;
6    } elsif ( /G/  ) {
7      ++$count_of_G;
8    } elsif ( /T/  ) {
9      ++$count_of_T;
10   } else {
11     print "!!!!!!!! Error - I don\'t recognize
     this base: ";
12     print;
13     print "\n";
14     ++$errors;
15   }
16 }
```

```
1  foreach $_ (@DNA) {
2    if ( $_ =~ /A/ ) {
3      ++$count_of_A;
4    } elsif ( $_ =~ /C/  ) {
5      ++$count_of_C;
6    } elsif ( $_ =~ /G/  ) {
7      ++$count_of_G;
8    } elsif ( $_ =~ /T/  ) {
9      ++$count_of_T;
10   } else {
11     print "!!!!!!!! Error - I don\'t recognize
     this base: ";
12     print $_;
13     print "\n";
14     ++$errors;
15   }
16 }
```

```perl
while ( <$FH> ) {
#while ( <> ) { # 处理命令行中指定的文件
   chomp;
   if ( /pattern/ ) {
      @fields = split /\t/;
    #@fields = split; # 以空格为分隔符
      ...
   }
   ...
}
```

Perl predefined variables

`perldoc perlvar`

```
1  while ( <$FH> ) {
2  #while ( <> ) { # 处理命令行中指定的文件
3    chomp;
4    if ( /pattern/ ) {
5      @fields = split /\t/;
6     #@fields = split; # 以空格为分隔符
7      ...
8    }
9    ...
10 }
```

Perl predefined variables

`perldoc perlvar`

```perl
# 在Perl中有四种方法给一个数字加1

# 方法一
++$count;

# 方法二（常用）
$count++;

# 方法三
$count = $count + 1;

# 方法四（递增值不是1时常用）
$count += 1;
```

```
1  $i = 0;
2  $j = 0;
3
4  # 先返回值，后加1
5  print $i++;   # 0
6
7  # 先加1，后返回值
8  print ++$j;   # 1
9
10 # --（减1）用法同++（加1）
```

双目赋值操作符

几乎所有用来求值的双目操作符都可以接上等号，成为相应的双目赋值操作符（binary assignment operator）。

```
1  $num = 9;
2  $num += 1;   # $num = $num + 1;   #10
3  $num -= 1;   # $num = $num - 1;   #9
4  $num *= 2;   # $num = $num * 2;   #18
5  $num /= 2;   # $num = $num / 2;   #9
6  $num **= 2;  # $num = $num ** 2;  #81
7  $num %= 2;   # $num = $num % 2;   #1
8
9  $str = "2";
10 $str .= "nd"; # $str = $str . "nd"; #2nd
11 $str x= 2;    # $str = $str x 2;    #2nd2nd
```

双目赋值操作符

几乎所有用来求值的双目操作符都可以接上等号，成为相应的双目赋值操作符（binary assignment operator）。

```
1  $num = 9;
2  $num += 1;   # $num = $num + 1;   #10
3  $num -= 1;   # $num = $num - 1;   #9
4  $num *= 2;   # $num = $num * 2;   #18
5  $num /= 2;   # $num = $num / 2;   #9
6  $num **= 2;  # $num = $num ** 2;  #81
7  $num %= 2;   # $num = $num % 2;   #1
8
9  $str = "2";
10 $str .= "nd"; # $str = $str . "nd"; #2nd
11 $str x= 2;    # $str = $str x 2;    #2nd2nd
```

- 大的字符串、数组都会占用计算机的大量内存
- 把字符串拆解成数组时，原始的字符串依然存在
- 大的字符串加上大的数组可能会导致计算机内存不足
- 计算机内存不足时，你比计算机先崩溃……

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

```
1 read in the DNA from a file
2
3 join the lines of the file into a single
    string of $DNA
4
5 # initialize the counts
6 count_of_A = 0
7 count_of_C = 0
8 count_of_G = 0
9 count_of_T = 0
```

```
for each base at each position in $DNA

  if base is A
    count_of_A = count_of_A + 1
  if base is C
    count_of_C = count_of_C + 1
  if base is G
    count_of_G = count_of_G + 1
  if base is T
    count_of_T = count_of_T + 1
done

print count_of_A, count_of_C, count_of_G,
  count_of_T
```

```
1 #!/usr/bin/perl -w
2 # Example 5-6   Determining frequency of
    nucleotides, take 2
3
4 # Get the DNA sequence data
5 print "Please type the filename of the DNA
    sequence data: ";
6
7 $dna_filename = <STDIN>;
8
9 chomp $dna_filename;
```

```
11 # Does the file exist?
12 unless ( -e $dna_filename ) {
13
14     print "File \"$dna_filename\" doesn\'t seem to exist
     !!\n";
15     exit;
16 }
17
18 # Can we open the file?
19 unless ( open( DNAFILE, $dna_filename ) ) {
20
21     print "Cannot open file \"$dna_filename\"\n\n";
22     exit;
23 }
24
25 @DNA = <DNAFILE>;
26
27 close DNAFILE;
```

```
29 $DNA = join( '', @DNA );
30
31 # Remove whitespace
32 $DNA =~ s/\s//g;
33
34 # Initialize the counts.
35 # Notice that we can use scalar variables to hold
     numbers.
36 $count_of_A = 0;
37 $count_of_C = 0;
38 $count_of_G = 0;
39 $count_of_T = 0;
40 $errors     = 0;
41
42 # In a loop, look at each base in turn, determine which
     of the
43 # four types of nucleotides it is, and increment the
44 # appropriate count.
```

```
45 for ( $position = 0 ; $position < length $DNA ; ++$position ) {
46
47     $base = substr( $DNA, $position, 1 );
48
49     if ( $base eq 'A' ) {
50         ++$count_of_A;
51     }
52     elsif ( $base eq 'C' ) {
53         ++$count_of_C;
54     }
55     elsif ( $base eq 'G' ) {
56         ++$count_of_G;
57     }
58     elsif ( $base eq 'T' ) {
59         ++$count_of_T;
60     }
61     else {
62         print "!!!!!!!! Error - I don\'t recognize this base:
   $base\n";
63         ++$errors;
64     }
65 }
```

```
67 # print the results
68 print "A = $count_of_A\n";
69 print "C = $count_of_C\n";
70 print "G = $count_of_G\n";
71 print "T = $count_of_T\n";
72 print "errors = $errors\n";
73
74 # exit the program
75 exit;
```

```
1 Please type the filename of the DNA sequence
    data: small.dna
2 !!!!!!!! Error - I don't recognize this vase:
     V
3 A = 40
4 C = 27
5 G = 24
6 T = 17
7 errors = 1
```

```
1 unless ( -e $dna_filename ) {
```

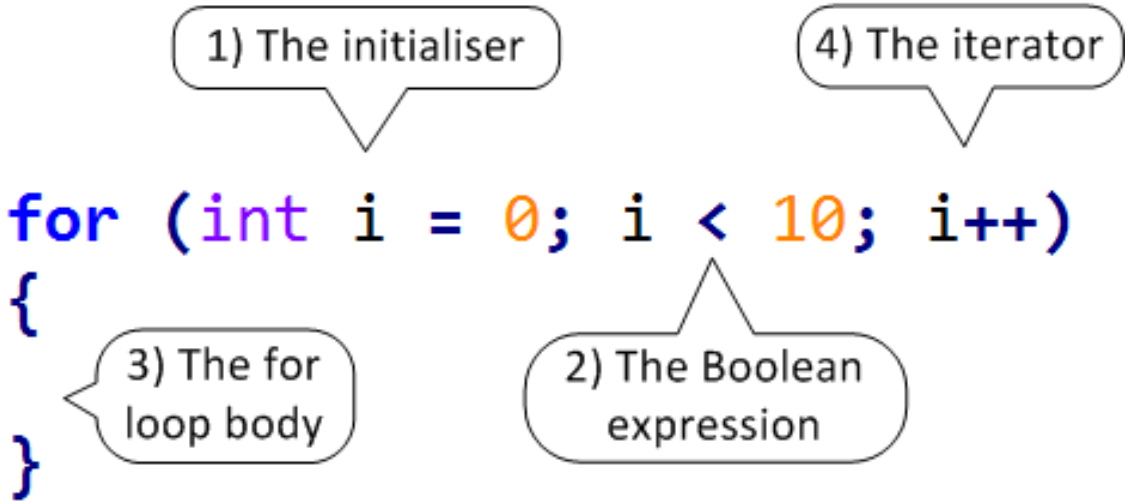
说明

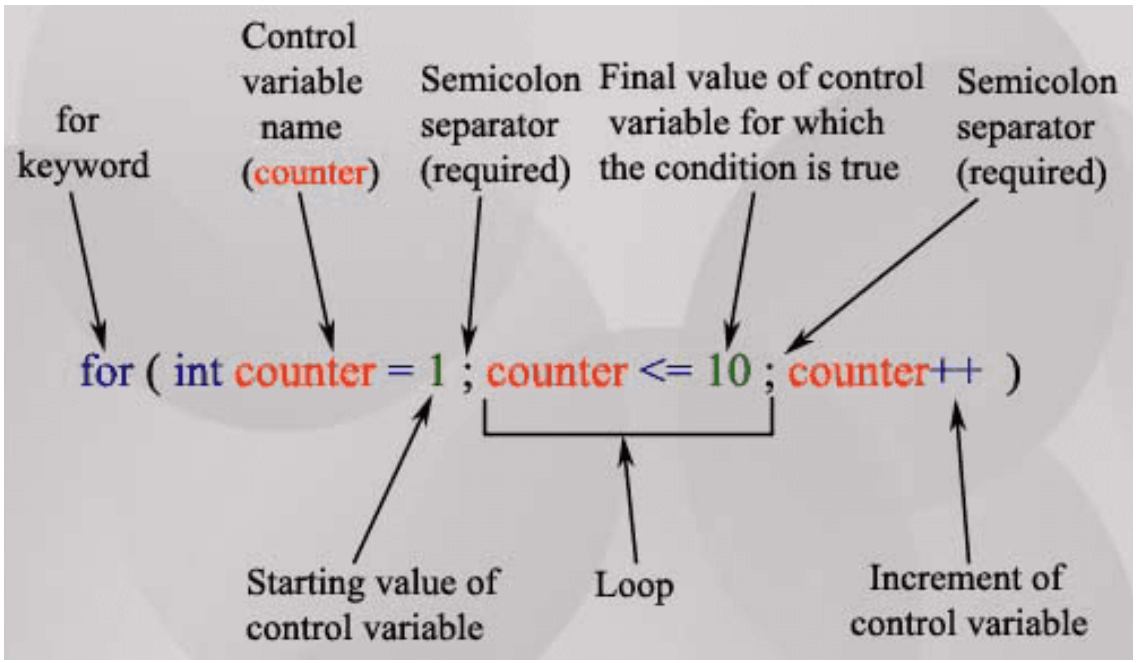
- 作用：测试文件是否存在（exist）
- 手册：`perldoc -f -X`（文件测试）

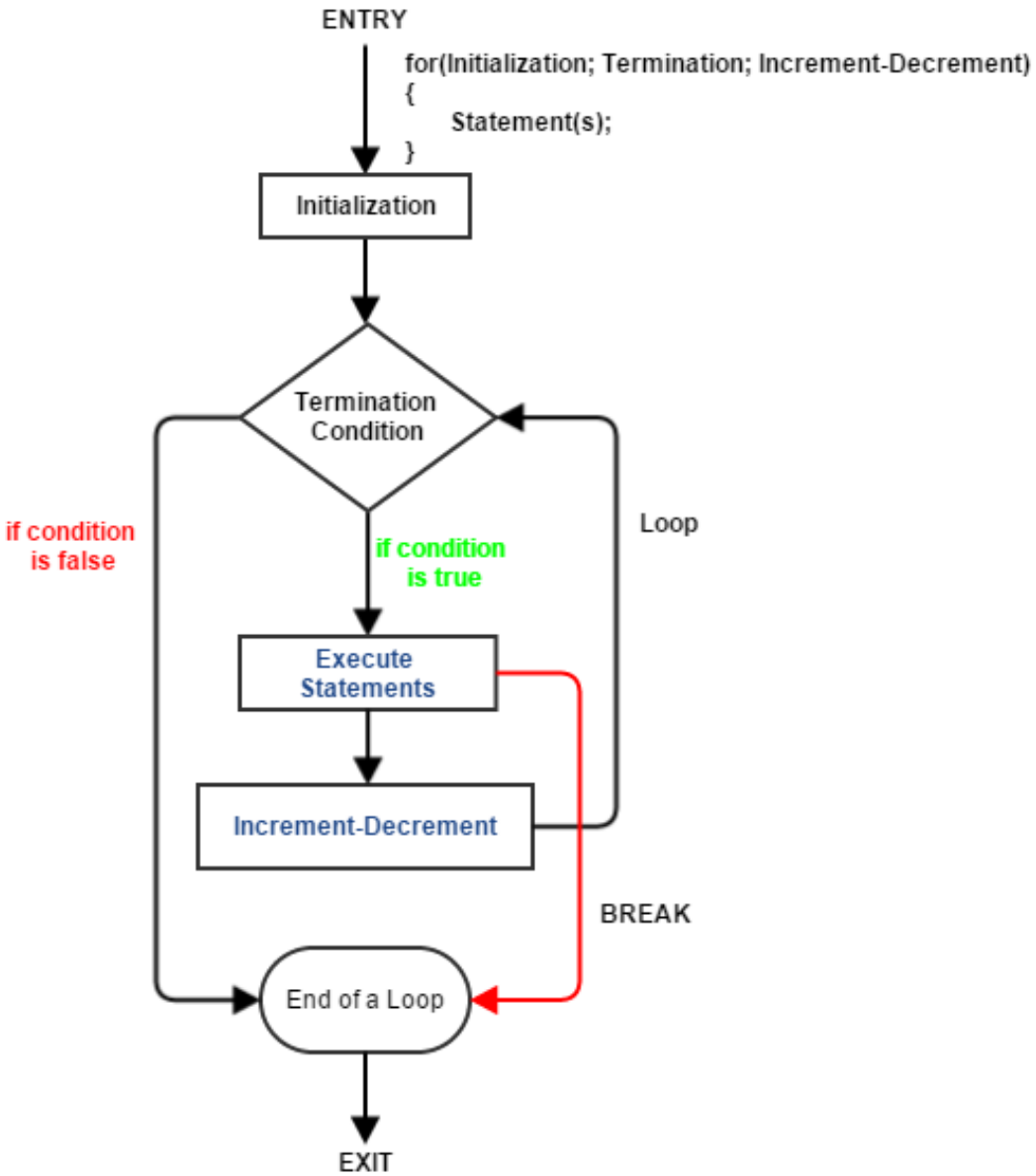
```
1 unless ( -e $dna_filename ) {
```

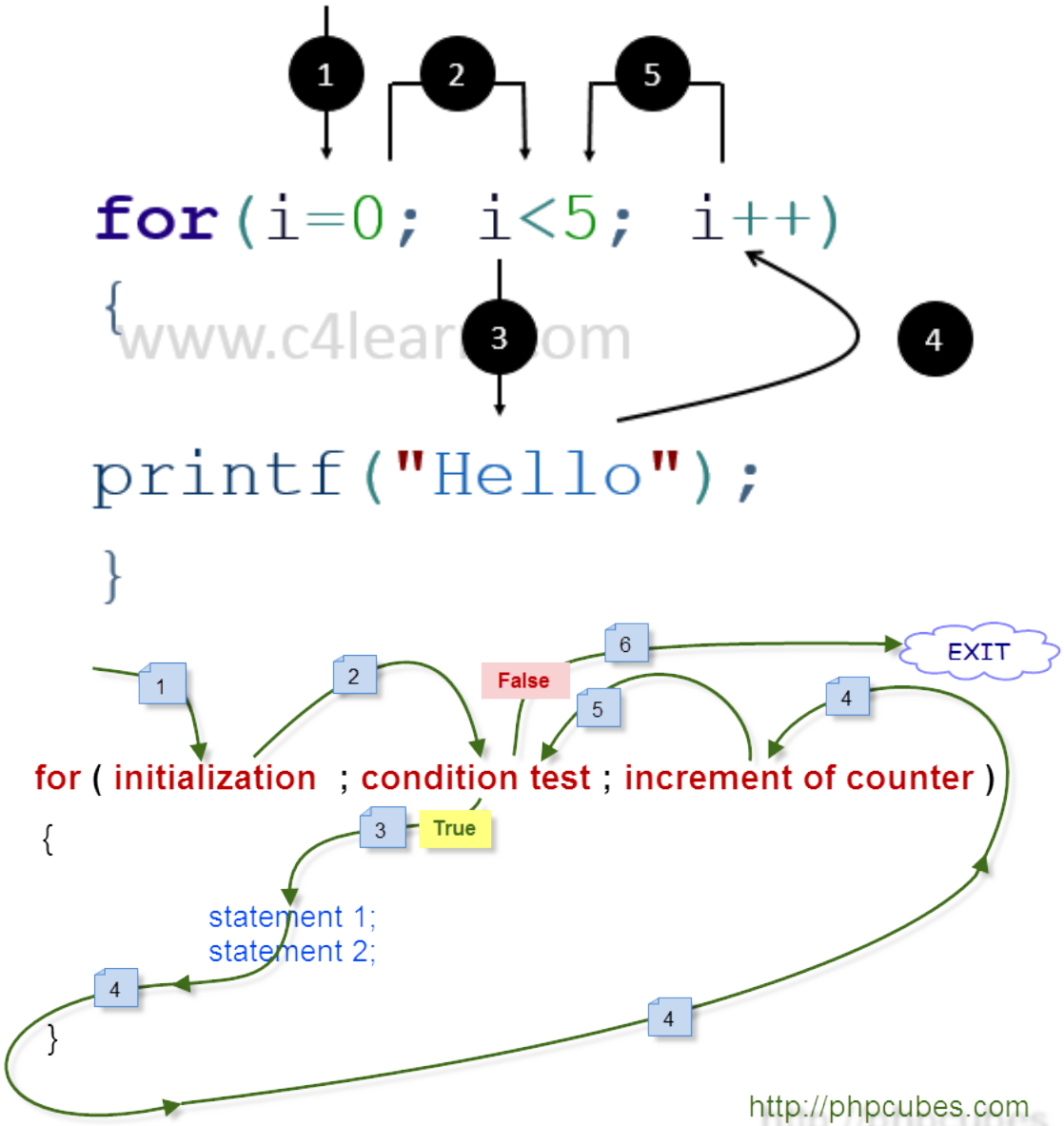
说明

- 作用：测试文件是否存在（exist）
- 手册：`perldoc -f -X`（文件测试）


1) The initialiser
4) The iterator
for (int i = 0; i < 10; i++)
{
3) The for loop body
2) The Boolean expression
}


for keyword
Control variable name (counter)
Semicolon separator (required)
Final value of control variable for which the condition is true
Semicolon separator (required)
for (int counter = 1 ; counter <= 10 ; counter++)
Starting value of control variable
Loop
Increment of control variable


ENTRY
for(Initialization; Termination; Increment-Decrement)
{
Statement(s);
}
Initialization
Termination Condition
if condition is false
if condition is true
Loop
Execute Statements
Increment-Decrement
BREAK
End of a Loop
EXIT


1
2
5
for(i=0; i<5; i++)
{
3
4
printf("Hello");
}
1
2
6
EXIT
False
5
4
for (initialization ; condition test ; increment of counter)
{
3
True
statement 1;
statement 2;
4
}
4
http://phpcubes.com

```
1 for ( $position = 0 ; $position < length $DNA
    ; ++$position ) {
2   # the statements in the block
3 }
```

```
1 # 等价的while循环
2 $position = 0;
3
4 while( $position < length $DNA  ) {
5   # the same statements in the block, plus
    ...
6
7   ++$position;
8 }
```

```
1 for ( $position = 0 ; $position < length $DNA
    ; ++$position ) {
2   # the statements in the block
3 }
```

```
1 # 等价的while循环
2 $position = 0;
3
4 while( $position < length $DNA  ) {
5   # the same statements in the block, plus
    ...
6
7   ++$position;
8 }
```

```perl
#!/usr/bin/perl

use strict;
use warnings;
use utf8;

for ( my $i = 1 ; $i <= 9 ; $i++ ) {
    for ( my $j = 1 ; $j <= $i ; $j++ ) {
        print "$j x $i = " . $i * $j;
        if ( $j == $i ) {
            print "\n";
        }
        else {
            print "\t";
        }
    }
}
```

```perl
#!/usr/bin/perl
use strict; use warnings; use utf8;

my $i = 1;
while ( $i <= 9 ) {
    my $j = 1;
    while ( $j <= $i ) {
        print "$j x $i = " . $i * $j;
        if ( $j == $i ) {
            print "\n";
        }
        else {
            print "\t";
        }
        $j++;
    }
    $i++;
}
```

```perl
#!/usr/bin/perl

use strict;
use warnings;
use utf8;

# for的语法上没有错误!
# 但输出的不是九九乘法表!
for ( my $i = 1, my $j = 1 ; $i <= 9 && $j <= $i ;
    $i++, $j++ ) {
      print "$j x $i = " . $i * $j;
      if ( $j == $i ) {
          print "\n";
      }
      else {
          print "\t";
      }
}
```

```perl
#!/usr/bin/perl
use strict; use warnings; use utf8;

my $i = 1;
my $j = 1;
while ( $i <= 9 ) {
    while ( $j <= $i ) {
        print "$j x $i = " . $i * $j;
        if ( $j == $i ) {
            print "\n";
        }
        else {
            print "\t";
        }
        $j++;
    }
    $i++;
}
```

索引

- 字符串：起始于 0，最后一个字符的索引比字符串的长度小 1
- 数组元素：第一个元素的索引为 0，最后一个元素的索引为 `scalar(@array)-1`，或者 `@array-1`

```
1 字符串: ATGCGCAT
2 索引值: 01234567
3 计数值: 12345678
4
5 数组元素: A C G T G T A C
6 元素索引: 0 1 2 3 4 5 6 7
7 元素计数: 1 2 3 4 5 6 7 8
```

索引

- 字符串：起始于 0，最后一个字符的索引比字符串的长度小 1
- 数组元素：第一个元素的索引为 0，最后一个元素的索引为 `scalar(@array)-1`，或者 `@array-1`

```
1 字符串: ATGCGCAT
2 索引值: 01234567
3 计数值: 12345678
4
5 数组元素: A C G T G T A C
6 元素索引: 0 1 2 3 4 5 6 7
7 元素计数: 1 2 3 4 5 6 7 8
```

```
1 $base = substr($DNA, $position, 1);
```

substr

- 作用：获取位置索引为 `$position` 的那个碱基
- substr 可以对字符串进行插入或者删除操作
- 第一个参数指定要操作的字符串
- 第二个参数指定要操作的位置索引（负值表示从字符串末尾开始）
- 第三个参数指定要操作的长度（负值表示字符串末尾剩余的字符数）
- 第四个参数指定要替换成的字符串
- substr vs. splice （操作字符串 vs. 操作数组）
  - 语法
  - 返回值
  - 对原数组的“伤害”

```
1 $base = substr($DNA, $position, 1);
```

substr

- 作用：获取位置索引为 `$position` 的那个碱基
- substr 可以对字符串进行插入或者删除操作
- 第一个参数指定要操作的字符串
- 第二个参数指定要操作的位置索引（负值表示从字符串末尾开始）
- 第三个参数指定要操作的长度（负值表示字符串末尾剩余的字符数）
- 第四个参数指定要替换成的字符串
- substr vs. splice （操作字符串 vs. 操作数组）
  - 语法
  - 返回值
  - 对原数组的“伤害”

```
1 my $s = "The black cat climbed the green tree
    ";
2 my $color  = substr $s, 4, 5;       # black
3 my $middle = substr $s, 4, -11; # black cat
    climbed the
4 my $end    = substr $s, 14; # climbed the
    green tree
5 my $tail   = substr $s, -4; # tree
6 my $z      = substr $s, -4, 2;      # tr
7
8 my $r = substr $s, 14, 7, "jumped from";    #
     climbed
9 # $s is now "The black cat jumped from the
    green tree"
```

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

```
1  # Also write the results to a file called "
     countbase"
2
3  $outputfile = "countbase";
4
5  unless ( open(COUNTBASE, ">$outputfile") ) {
6    print "Cannot open file \"$outputfile\" to
     write to!!\n\n";
7    exit;
8  }
9
10 print COUNTBASE "A=$a C=$c G=$g T=$t errors=
     $e\n";
11
12 close(COUNTBASE);
```

```
1  # 读取文件
2  open my $FH, '<', $filename or die "$0 : failed to
       open input file '$filename' : $!\n";
3  ... <$FH> ...
4  close $FH or warn "$0 : failed to close input file
       '$filename' : $!\n";
5
6  # 写入文件
7  open my $FH_OUT, '>', $fn_out or die "$0 : failed
     to open output file '$fn_out' : $!\n";
8  select $FH_OUT;
9  print "something...";
10 # OR: use $FH_OUT for every print
11 #print $FH_OUT "something...";
12 ...
13 close $FH_OUT or warn "$0 : failed to close output
       file '$fn_out' : $!\n";
```

```
1  #!/usr/bin/perl -w
2  # Example 5-7   Determining frequency of nucleotides,
     take 3
3
4  # Get the DNA sequence data
5  print "Please type the filename of the DNA sequence data
     : ";
6
7  $dna_filename = <STDIN>;
8
9  chomp $dna_filename;
10
11 # Does the file exist?
12 unless ( -e $dna_filename ) {
13
14     print "File \"$dna_filename\" doesn\'t seem to exist
     !!\n";
15     exit;
16 }
```

```
18 # Can we open the file?
19 unless ( open( DNAFILE, $dna_filename ) ) {
20
21     print "Cannot open file \"$dna_filename\"\n\n"
   ;
22     exit;
23 }
24
25 @DNA = <DNAFILE>;
26
27 close DNAFILE;
28
29 $DNA = join( '', @DNA );
30
31 # Remove whitespace
32 $DNA =~ s/\s//g;
```

```
34 # Initialize the counts.
35 # Notice that we can use scalar variables to
     hold numbers.
36 $a = 0;
37 $c = 0;
38 $g = 0;
39 $t = 0;
40 $e = 0;
41
42 # Use a regular expression "trick", and five
     while loops,
43 #  to find the counts of the four bases plus
     errors
```

```
44 while ( $DNA =~ /a/ig )          { $a++ }
45 while ( $DNA =~ /c/ig )          { $c++ }
46 while ( $DNA =~ /g/ig )          { $g++ }
47 while ( $DNA =~ /t/ig )          { $t++ }
48 while ( $DNA =~ /[^acgt]/ig ) { $e++ }
49
50 print "A=$a C=$c G=$g T=$t errors=$e\n";
```

```
52 # Also write the results to a file called "
     countbase"
53 $outputfile = "countbase";
54
55 unless ( open( COUNTBASE, ">$outputfile" ) ) {
56
57     print "Cannot open file \"$outputfile\" to
     write to!!\n\n";
58     exit;
59 }
60
61 print COUNTBASE "A=$a C=$c G=$g T=$t errors=$e\n";
62
63 close(COUNTBASE);
64
65 # exit the program
66 exit;
```

```
1 Please type the filename of the DNA sequence
    data: small.dna
2 A=40 C=27 G=24 T=17 errors=1
```

```
1 while ( $dna =~ /a/ig ) { $a++ }
```

while 循环

- i：不区分大小写（匹配 a 或者 A）
- g：全局修饰符，匹配字符串中的所有 a
- 没有 g 的话，如果字符串中有 a，会陷入死循环
- 每一次匹配都会递增计数器（对字符串中的所有 a 进行计数）

```
1 while ( $dna =~ /a/ig ) { $a++ }
```

while 循环

- i：不区分大小写（匹配 a 或者 A）
- g：全局修饰符，匹配字符串中的所有 a
- 没有 g 的话，如果字符串中有 a，会陷入死循环
- 每一次匹配都会递增计数器（对字符串中的所有 a 进行计数）

```
1 $a = ($dna =~ tr/Aa//);
2 $c = ($dna =~ tr/Cc//);
3 $g = ($dna =~ tr/Gg//);
4 $t = ($dna =~ tr/Tt//);
5
6 $basecount = ($dna =~ tr/ACGTacgt//);
7 $nonbase = (length $dna) - $basecount;
```

tr

- tr 函数返回它在字符串中找到的特定字符的数目
- 如果替换的字符集为空，原始的字符串不会发生改变
- 速度快，一个很好的字符计数器
- 需要同时指定大小写字母
- tr 不接受字符集（没法直接对非碱基的字符进行计数）

```
1 $a = ($dna =~ tr/Aa//);
2 $c = ($dna =~ tr/Cc//);
3 $g = ($dna =~ tr/Gg//);
4 $t = ($dna =~ tr/Tt//);
5
6 $basecount = ($dna =~ tr/ACGTacgt//);
7 $nonbase = (length $dna) - $basecount;
```

tr

- tr 函数返回它在字符串中找到的特定字符的数目
- 如果替换的字符集为空，原始的字符串不会发生改变
- 速度快，一个很好的字符计数器
- 需要同时指定大小写字母
- tr 不接受字符集（没法直接对非碱基的字符进行计数）

```perl
my $dna = "ACGT";
my $dna_len = length $dna;
print "$dna_len"; # 4

my @bases = ("A", "C", "G", "T");
my $bases_len = length @bases; # NOT USE!
print "$bases_len"; # 1
#Warning: length() used on @bases (did you
  mean "scalar(@bases)"?)
```

length

Returns the length in characters of the value of EXPR. If EXPR is omitted, returns the length of `$_`. If EXPR is undefined, returns “undef”.

This function cannot be used on an entire array or hash to find out how many elements these have. For that, use “scalar @array” and “scalar keys %hash”, respectively.

```perl
my $dna = "ACGT";
my $dna_len = length $dna;
print "$dna_len"; # 4

my @bases = ("A", "C", "G", "T");
my $bases_len = length @bases; # NOT USE!
print "$bases_len"; # 1
#Warning: length() used on @bases (did you
  mean "scalar(@bases)"?)
```

length

Returns the length in characters of the value of EXPR. If EXPR is omitted, returns the length of `$_`. If EXPR is undefined, returns “undef”.

This function cannot be used on an entire array or hash to find out how many elements these have. For that, use “scalar @array” and “scalar keys %hash”, respectively.

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

Metacharacters are special characters

\	Quote the next metacharacter
^	Match the beginning of the line
.	Match any character (except newline)
$	Match the end of the line (or before newline at the end)
|	Alternation
()	Grouping
[]	Character class

Quantifiers: how many?

*	Match 0 or more times
+	Match 1 or more times
?	Match 1 or 0 times
{n}	Match exactly n times
{n,}	Match at least n times
{n,m}	Match between n and m times (n,m <=32766)

More on Character Classes:

^	As the first char, negates the class ([^A] matches anything except "A") Otherwise literal caret.
-	between 2 chars is a range ([A-G] will match A,B,C,D,E,F, or G) otherwise literal dash
\b	backspace (easiest way to get one)
]	as first char, literal close bracket

\w\W\s\S\d\D can all be used in a character class (but not in a range)

Regex Pattern Shortcuts

\w	Match a "word" character (alphanumeric plus "_")
\W	Match a non-"word" character
\s	Match a whitespace character
\S	Match a non-whitespace character
\d	Match a digit character
\D	Match a non-digit character

Stuff That's Hard to Type

\t	tab
\n	newline
\r	return (carriage return)
\f	form feed
\e	escape
\033	octal char
\x1B	hex char
\c[	control char

These don't take up any space:

\b	Match a word boundary
\B	Match a non-(word boundary)
\A	Match only at beginning of string
\Z	Match only at end of string, or before newline at the end

Change how the pattern works:

```
/i    ignore case
/g    return all matches
/s    period (.) now matches newline (\n)
/m    "multi-line": ^ and $ match near
        any \n, not just start/end of string
/x    "expanded" regex: space and newlines
        are ignored, and # comments allowed:
       Example:
          /\w+          # first name
           \s+          # space
           \d+          # age
          /x
```

元 字 符	含义描述
\d	匹配任意一个十进制数字，等价于[0-9]
\D	匹配任意一个除十进制数字以外的字符，等价于[^0-9]
\s	匹配任意一个空白字符，等价于[\f\n\r\t\v]
\S	匹配除空白字符以外任何一个字符，等价于[^\f\n\r\t\v]
\w	匹配任意一个数字、字母或下画线，等价于[0-9a-zA-Z_]
\W	匹配除数字、字母或下画线以外的任意一个字符，等价于[^0-9a-zA-Z_]
*	匹配 0 次、1 次或多次其前的原子
+	匹配 1 次或多次其前的原子
?	匹配 0 次或 1 次其前的原子
.	匹配除了换行符外的任意一个字符
|	匹配两个或多个分支选择
{n}	表示其前面的原子恰好出现 n 次
{n, }	表示其前面的原子出现不少于 n 次
{n, m}	表示其前面的原子至少出现 n 次，最多出现 m 次
^或\A	匹配输入字符串的开始位置（或在多行模式下行的开头，即紧随一换行符之后）
$或\Z	匹配输入字符串的结束位置（或在多行模式下行的结尾，即紧随一换行符之前）
\b	匹配单词的边界
\B	匹配除单词边界以外的部分
[]	匹配方括号中指定的任意一个原子
[^]	匹配除方括号中的原子以外的任意一个字符
()	匹配其整体为一个原子，即模式单元。可以理解为由多个单个原子组成的大原子

Atoms		Quantifiers	
Plain symbol:	`...`	Universal quantifier:	`*`
Escape:	`\`	Non-greedy universal quantifier:	`*?`
Grouping operators:	`( )`	Existential quantifier:	`+`
Backreference:	`\#,\##`	Non-greedy existential quantifier:	`+?`
Character class:	`[ ]`	Potentiality quantifier:	`?`
Digit character class:	`\d`	Non-greedy potentiality quantifier:	`??`
Non-digit character class:	`\D`	Exact numeric quantifier:	`{num}`
Alphanumeric char class:	`\w`	Lower-bound quantifier:	`{min,}`
Non-alphanum char class:	`\W`	Bounded numeric quantifier:	`{min,max}`
Whitespace char class:	`\s`	Non-greedy bounded quantifier:	`{min,max}?`
Non-whitespace char class	`\S`		
Wildcard character:	`.`	**Group-Like Patterns**	
Beginning of line:	`^`	Pattern modifiers:	`(?Limsux)`
Beginning of string:	`\A`	Comments:	`(?#...)`
End of line:	`$`	Non-backreferenced atom:	`(?:...)`
End of string:	`\Z`	Positive Lookahead assertion:	`(?=...)`
Word boundary:	`\b`	Negative Lookahead assertion:	`(?!...)`
Non-word boundary:	`\B`	Positive Lookbehind assertion:	`(?<=...)`
Alternation operator:	`|`	Negative Lookbehind assertion:	`(?<!...)`
		Named group identifier:	`(?P<name>)`
Constants		Named group backreference:	`(?P=name)`

Perl Regular Expression Quick Reference 1.05

N.B.: this quick reference is just that - some of the explanations have been simplified. For the authoritative documentation, see the latest edition of *Programming Perl* or `perldoc perlre`.

Specific characters:

`\t`	A tab character
`\n`	A newline character (OS neutral)
`\r`	A carriage return character
`\f`	A form feed character
`\cX`	Control character CTRL-*X*
`\NNN`	Octal code for character *NNN*

Metacharacters:

The following 12 characters need to be escaped with a backslash - "\" - because by default, they mean something special.

`\ | ( ) [ { ^ $ * + ? .`

`.`	Match any one character (except `\n`)
`|`	Alternation
`( )`	Group and capture
`[ ]`	Define character class
`\`	Modify the meaning of the next char.

Anchors:

`^`	Match at the beginning of a string (or line)
`$`	Match at the end of a string (or line)
`\b`	Match at a 'word' boundary
`\B`	Match at not a 'word' boundary

These are also known as *zero width assertions*.

Quantifiers:

These quantifiers apply to the preceding *atom*.

`*`	Match 0 or more times
`+`	Match 1 or more times
`?`	Match 0 or 1 times
`{N}`	Match exactly *N* times
`{N,}`	Match at least *N* times
`{N,M}`	Match at least *N* but not more than *M* times

By default, quantifiers are "*greedy*". They attempt to match as many characters as possible. In order to make them match as few characters as possible, follow them with a question mark "?".

Character class metacharacters:

`^`	If the first character of a class, negates that class
`-`	Unless first or last character of a class, used for a range

Character class shortcuts:

`\d`	`[0-9]`	A digit
`\D`	`[^0-9]`	A non-digit
`\s`	`[ \t\n\r\f]`	A whitespace char.
`\S`	`[^ \t\n\r\f]`	A non-whitespace char.
`\w`	`[a-zA-Z0-9_]`	A 'word' char.
`\W`	`[^a-zA-Z0-9_]`	A 'non-word' char.

These shortcuts can be used either on their own, or within a character class.

Metaquote & case translations:

`\Q`	Quote (de-meta) characters until `\E`
`\U`	Uppercase characters until `\E`
`\L`	Lowercase characters until `\E`

Special variables:

`$``	The characters to the left of the match
`$&`	The characters that matched
`$'`	The characters to the right of the match
`\N`	The characters captured by the *N*th set of parentheses (if on the match side)
`$N`	The characters captured by the *N*th set of parentheses (if not on the match side)

Modifiers:

These modifiers apply to the entire pattern

`/i`	Ignore case
`/g`	Match globally (all)
`/m`	Let `^` and `$` match next to embedded `\n`
`/s`	Let `.` match `\n`
`/x`	Ignore most whitespace and allow comments
`/e`	Evaluate right hand side of `s///` as an expression

All except `/e` apply to both `m//` and `s///`.

Binding operators:

`=~`	True if the regex matches
`!~`	True if the regex doesn't match

expression	matches...
abc	abc (that exact character sequence, but anywhere in the string)
^abc	abc at the *beginning* of the string
abc$	abc at the *end* of the string
a|b	either of a and b
^abc|abc$	the string abc at the beginning or at the end of the string
ab{2,4}c	an a followed by two, three or four b's followed by a c
ab{2,}c	an a followed by at least two b's followed by a c
ab*c	an a followed by any number (zero or more) of b's followed by a c
ab+c	an a followed by one or more b's followed by a c
ab?c	an a followed by an optional b followed by a c; that is, either abc or ac
a.c	an a followed by any single character (not newline) followed by a c
a\.c	a.c exactly
[abc]	any one of a, b and c
[Aa]bc	either of Abc and abc
[abc]+	any (nonempty) string of a's, b's and c's (such as a, abba, acbabcacaa)
[^abc]+	any (nonempty) string which does *not* contain any of a, b and c (such as defg)
\d\d	any two decimal digits, such as 42; same as \d{2}
\w+	a "word": a nonempty sequence of alphanumeric characters and low lines (underscores), such as foo and 12bar8 and foo_1
100\s*mk	the strings 100 and mk optionally separated by any amount of white space (spaces, tabs, newlines)
abc\b	abc when followed by a word boundary (e.g. in abc! but not in abcd)
perl\B	perl when *not* followed by a word boundary (e.g. in perlert but not in perl stuff)

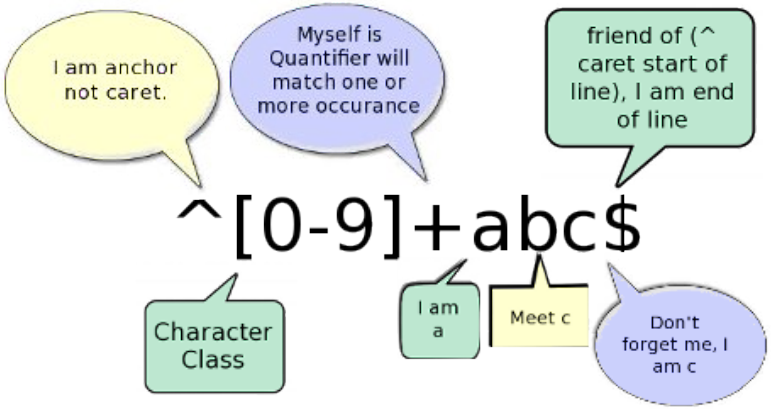
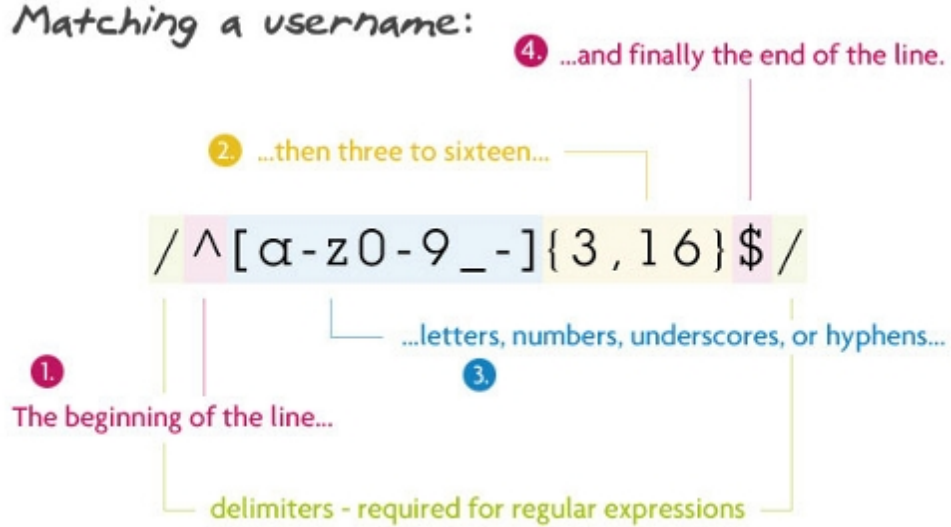
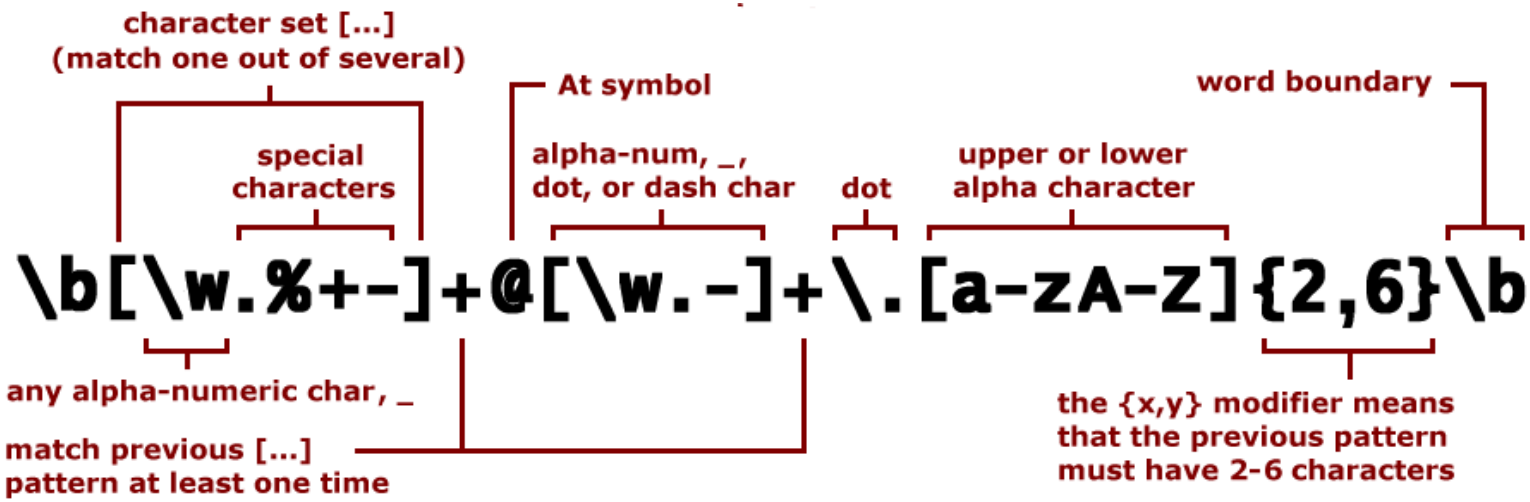


Illustration of Regular Expression





Parse: username@domain.TLD (top level domain)

Regexp::Common

Provide commonly requested regular expressions.

```
use Regexp::Common;
while (<>) {
    /$RE{num}{real}/                    and print q{a number};
    /$RE{quoted}/                       and print q{a ['"`] quoted
  string};
    m[$RE{delimited}{-delim=>'/'}]  and print q{a /.../ sequence
  };
    /$RE{balanced}{-parens=>'()'}/ and print q{balanced
  parentheses};
    /$RE{profanity}/                    and print q{a #*@%-ing word};
}
```

Regexp::Common::X

Regexp::Common::time, Regexp::Common::URI, Regexp::Common::Email::Address, ...

Basic Matching:

```
$var =~ m/something here/;
```

Capturing:

```
$var =~ m/(pattern)/;
```

$1 contains "pattern"

```
$catch = $1;
```

or

```
($catch) = $var =~ m/(pattern)/;
($x,$y) = $var =~ m/(first) (second)/;
```

Backreferences: refer to previous match

```
$var = "The red red book.";
$var =~ m/(red) \1/;
```

Basic Substitutions:

```
$var =~ s/pattern/replacement/;
```

"this" is any m/pattern/

"that" is a double-quoted string

For example

```
$var =~ s/(first) (last)/$2 $1/;
```

Backreferences belong in pattern only!

Inline captures don't work with s///:

```
($x) = $var =~ s/(this)/that/;   # wrong
```

Use $1 instead:

```
$x = $1;
```

Wacky Test Names

```
while (<>)
{ #              FREQ    BYP4      TRY
  if ( m/mpll_(\d+)_(byp|by4)_(\w)/i )
  {   $data{$die}{FREQ} = $1;
      $data{$die}{BYP4} = $2;
      $data{$die}{TRY}  = $3;
      next;
  }
}
```

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

知识点

- 流程控制的各种语句：条件判断，循环
- 文件交互：打开文件、读取数据、写入文件
- 标量和数组的转换：join，split
- 正则表达式：字符集，模式匹配
- 字符串操作：substr，tr
- 其他：真与假，获取键盘输入，变量初始化，智能化处理，特殊变量，递增，文件测试，索引

技能

- 能够熟练使用 Perl 语言中的各种流程控制语句
- 能够编写在 DNA 或者蛋白质序列中查找基序的程序
- 能够编写对 DNA 序列中的核苷酸进行计数的程序

知识点

- 流程控制的各种语句：条件判断，循环
- 文件交互：打开文件、读取数据、写入文件
- 标量和数组的转换：join，split
- 正则表达式：字符集，模式匹配
- 字符串操作：substr，tr
- 其他：真与假，获取键盘输入，变量初始化，智能化处理，特殊变量，递增，文件测试，索引

技能

- 能够熟练使用 Perl 语言中的各种流程控制语句
- 能够编写在 DNA 或者蛋白质序列中查找基序的程序
- 能够编写对 DNA 序列中的核苷酸进行计数的程序

教学提纲

1. 引言
2. 流程控制
   - 条件判断
   - 循环
3. 代码布局
4. 查找基序
   - 获取键盘输入
   - 数组变标量
   - do-until 循环
   - 正则表达式
   - 模式匹配
5. 计数核苷酸
   - 把字符串拆解成数组
   - 操作字符串
6. 写入文件
7. 知识拓展
8. 回顾与总结
   - 总结
   - 思考题

1. 总结 Perl 语言中进行流程控制的语句。
2. 总结 Perl 语言中真与假的判断法则。
3. 比较 chomp 和 chop。
4. 总结标量（字符串）和数组互转的方法。
5. 总结正则表达式和模式匹配的使用。
6. 如果不对变量初始化会怎样？
7. 用实例说明 Perl 对数字和字符串的智能化处理。
8. 总结 substr 和 tr 对字符串的处理。

下节预告

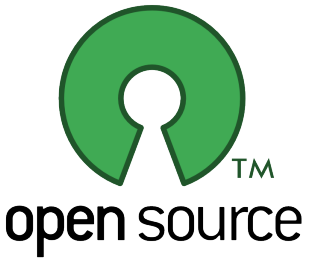
问题

计数核苷酸的程序有好几个，但其实每个版本的大部分代码都是一样的，唯一改变的是计数碱基的部分。有没有办法能够比较方便地只改变计数碱基的部分呢？

回顾

shell 编程中的函数（定义、调用、参数传递、作用域等）。

Powered by



TEX LATEX XETEX Beamer