1.0.1.8.6

Introduction to Perl
Session 6

- special variables
- subroutines

I/O Recap

- file handles are created using open(F,$file);
  - for reading “$file”
  - for writing “>$file”
  - for appending “>>$file”
- records are read from the filehandle using <F>

```
open(F,$file);
while($line = <F>) {
  chomp $line;
  ($a,$b,$c) = split(“ “,$line);
}
close(F);

open(F,”>$file”);
for $line (@lines) {
  @tokens = split(“ “, $line);
  printf F “%d %s\n”, @tokens;
}
close(F);
```

Special Variables

- Perl has a large number of special variables
  - special variables are *contextual* – store helpful values
  - special variables can *radically change the behaviour* of your code
  - special variables are used as *default inputs* to certain functions

- special variable names are generally unusual and the names do not adhere to naming rules of variables you can create
  - `$_`
  - `$,`
  - `$\`
  - `$1`

- special variables help you write more concise code

Special Variables - $a $b

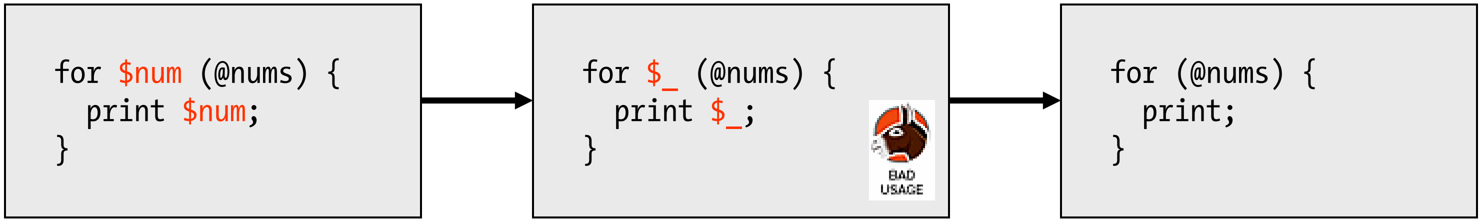
- we have already seen special variables $a and $b
  - “magic sauce” in sort code
  - do not need to be declared ahead of time
  - take on different values as code runs

```
sort { $b <=> $a } @nums;
```

- another special variable we saw was $#array
  - stored last index value of @array
  - could be used to explicitly shrink the array

Special Variables - $_

- the variable $_ is ubiquitous in Perl code, even when it is not explicitly mentioned
- it is the default input to many functions
- it holds the value in the current input, iteration or pattern search space



```
for $num (@nums) {
  print $num;
}
```

```
for $_ (@nums) {
  print $_;
}
```

```
for (@nums) {
  print;
}
```

- within a for loop, $_ points to the current value in the list
  - $_ is not a copy of the variable – it is an alias
  - assigning a value to $_ changes the value in the list

- without arguments, print will send the value of $_ to standard output

$_ in loops

```
for (1..5) {
  # $_ holds the value 1, 2, 3, etc as the loop iterates
  $x = rand();
  printf("number %d is %f",$_,$x);
}

number 1 is 0.945200
number 2 is 0.586325
...
```

- $_ points to the current iterator value of the immediate loop

$_ in nested loops

- $_ is the iterator value of inner-most loop in which it appears

```
for (1..3) {
  # printing value of $_ in (1..3) loop
  print;
  for (a..c) {
    # printing value of $_ in (a..c) loop
    print;
  }
}

1
a
b
c
2
a
b
c
3
a
b
c
```

$_ as default argument

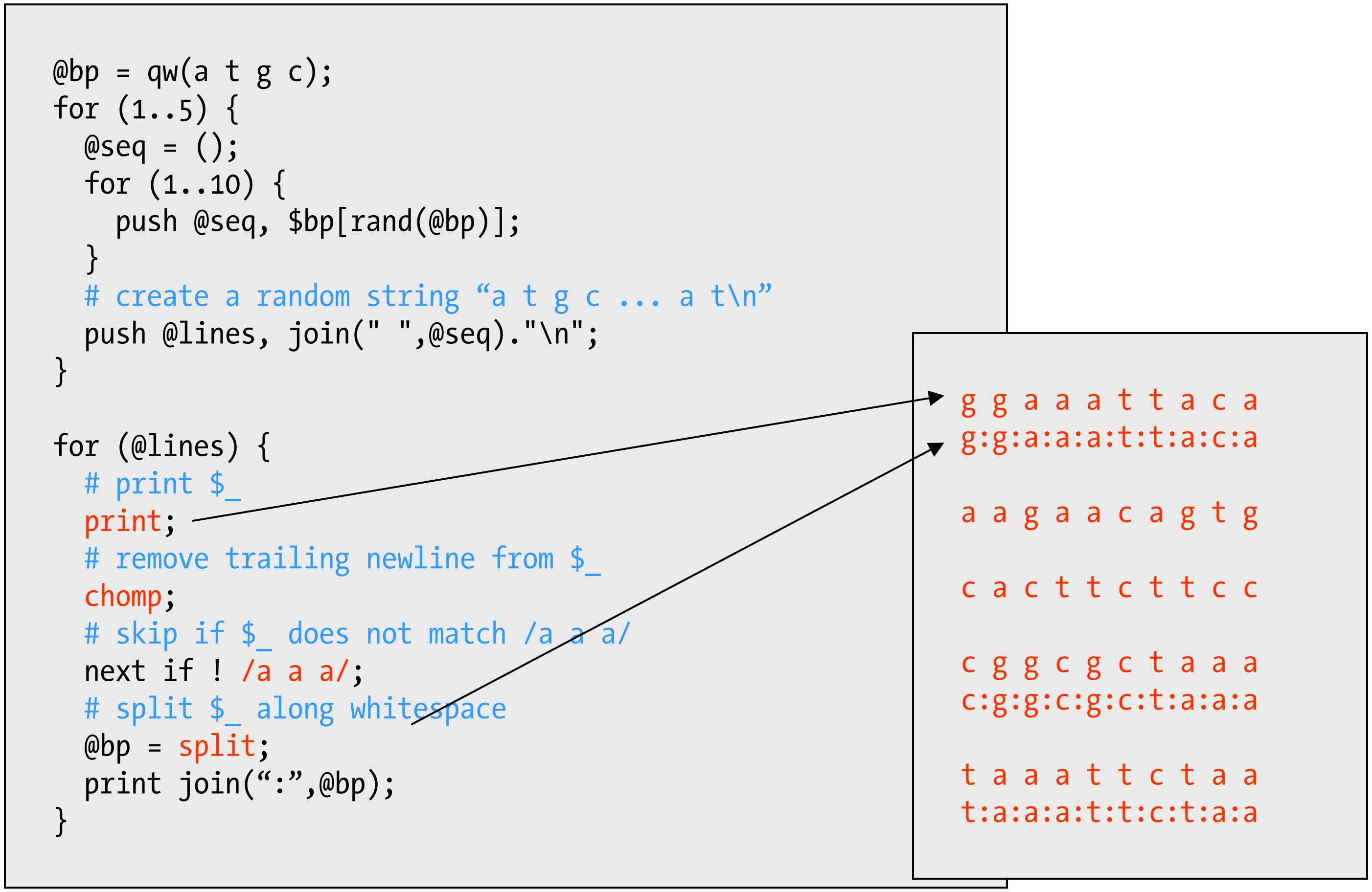
- we’ve seen that print without an argument prints $_
- unary operators like defined also test $_ if no arguments are passed

```
for (1..2,undef) {
  print if defined;
  for (a,undef,b) {
    print;
  }
}

1
a
      ← undef from (a,undef,b)
b
2
a
      ← undef from (a,undef,b)
b
a
      ← undef from (a,undef,b)
b
```

$_ as default argument

- chomp and split and m// also take $_ as default argument



```
@bp = qw(a t g c);
for (1..5) {
  @seq = ();
  for (1..10) {
    push @seq, $bp[rand(@bp)];
  }
  # create a random string “a t g c ... a t\n”
  push @lines, join(" ",@seq)."\n";
}

for (@lines) {
  # print $_
  print;
  # remove trailing newline from $_
  chomp;
  # skip if $_ does not match /a a a/
  next if ! /a a a/;
  # split $_ along whitespace
  @bp = split;
  print join(“:”,@bp);
}
```

```
g g a a a t t a c a
g:g:a:a:a:t:t:a:c:a

a a g a a c a g t g

c a c t t c t t c c

c g g c g c t a a a
c:g:g:c:g:c:t:a:a:a

t a a a t t c t a a
t:a:a:a:t:t:c:t:a:a
```

$_ for conciseness

- $_ helps you limit verbosity in your code
  - calling functions without arguments may feel strage, but the feeling will pass

```
for $line (@lines) {
  print $line;
  chomp $line;
  next if ! $line =~ /a a a/;
  @bp = split(“ “,$line);
}
```

→

```
for $_ (@lines) {
  print $_;
  chomp $_;
  next if ! $_ =~ /a a a/;
  @bp = split(“ “,$_);
}
```

BAD USAGE

→

```
for (@lines) {
  print;
  chomp;
  next if ! /a a a/;
  @bp = split;
}
```

- you should rarely need to explicitly refer to $_ in your code
  - `next if /abc/` instead of `next if $_ =~ /abc/`
  - `print` instead of `print $_`
  - `chomp` instead of `chomp $_`
- except in cases where no default argument is possible
  - `printf(“%s %d”,$string,$_);`
  - `$sum += $_;`

capital crimes with $_

- $_ is an alias not a copy
- you should never assign values to $_ under penalty of becoming a donkey

BAD USAGE

```
@nums = (1..3);
for (@nums) {
  print;
  # insidious
  $_ = 6;
}

for (@nums) {
  print;
}

1
2
3

6
6
6
```

- think of $_ as a pointer to the current iterated value
- if you change $_, you change the value
- for this reason, $_ is used as a read-only variable in vast majority of cases
- if you need to work with the value of $_ destructively, assign it to a variable
  - `$line = $_`

obfuscation with $_

- you can use the alias nature of $_ to alter a list
  - I strongly recommend you never do this

```
@nums = (1..3);

for (@nums) {
  $_++;
}
# nums now (2,3,4);

for (@nums) {
  $_ = sprintf("%d.%d",$_**2,$_);
}
# nums now (4.2,9.3,16.4);

for (@nums) {
  $_ = int;
}
# nums now (4,9,16);
```

Other Special Variables

- assignment
  - *man perlvar*
  - read about the following special variables
    - `$_`
    - `@_`
    - `$,`
    - `$”`
    - `$\`
    - `$/`
    - `$$`
    - `$.`
  - write a script that uses all these special variables

Introduction to Subroutines

- a subroutine is a named chunk of code you can call as a function
  - adds modularity to your scripts
  - helps you reuse code

```
@bp = qw(a t g c);
@lines = ();
for (1..5) {
  @seq = ();
  for (1..10) {
    push @seq, $bp[rand(@bp)];
  }
  # create a random string “a t g c ... a t”
  push @lines, join(" ",@seq);
}
```

```
@lines = ();
for (1..5) {
  # call the function, store output in $seq
  $seq = make_sequence();
  push @lines, $seq;
}

# create a random string “a t g c ... a t”
sub make_sequence {
  @bp = qw(a t g c);
  @seq = ();
  for (1..10) {
    push @seq, $bp[rand(@bp)];
  }
  $seq = join(“ “,@seq);
  return $seq;
}
```

Introduction to Subroutines

- you provide the name of the subroutine
  - make the name explicit and specific
    - `get_gc_ratio()` vs `process_sequence()`
    - `remove_vowels()` vs `munge_string()`
  - variety of naming conventions exist
    - getStringLength()
    - get_string_length()
    - e.g., imperative verb + (adjective) + noun
      - get_next_record()
      - store_current_state()
- subroutines generally return values via `return`
- always call subroutines with `()`, even if no arguments are passed

```
$x = NAME();
sub NAME {
  ...
  return $value;
}
```

Passing Arguments

- subroutines are most useful when they accept arguments that control their behaviour
- consider the subroutine below which creates a random 10mer
  - what about making an n-mer?

```
$seq = make_sequence();

# create a random 10-mer
# this is not a very reusable function
sub make_sequence {
  @bp = qw(a t g c);
  $seq = “”;
  for (1..10) {
    $seq .= $bp[rand(@bp)];
  }
  return $seq;
}
```

Passing Arguments

- a subroutine accepts a list as argument (one or more scalars)
- the special variable @_ within the subroutine is populated with aliases to the arguments
  - elements of @_ are $_[0], $_[1], $_[2], ...
  - just like $_, do not modify @_
  - modifying @_ changes the values of the original variables

```
mysub(1,2,3);

sub mysub {
  # arguments available via @_ special variable
  # assign to variables in one shot
  ($arg1,$arg2,$arg3) = @_;
  # or separately
  $arg1 = $_[0];
  $arg2 = $_[1];
  $arg3 = $_[3];
  return $arg1+$arg2+$arg3;
}
```

Passing Arguments

- upon receiving @_ in the function, it is customary to create a copy of the values to prevent inadvertent modification

```
print sum(1);
print sum(1,2,3);

# compute and return the sum of a list
sub sum {
  # explicitly make a copy of arguments
  @nums = @_;
  $sum = 0;
  for (@nums) {
    $sum += $_;
  }
  return $sum;
}
```

```
sub sum {
  $sum = 0;
  # iterating through @_ directly
  for (@_) {
    # $_ alias to each argument
    $sum += $_;
  }
  return $sum;
}
```

- in certain cases, if you're careful, you can traverse @_ directly
  - make sure what you are doing is going to be obvious to the reader
- in other cases, copying @_ is too costly and you need to work with aliases

Passing Arguments

- it is customary to create specifically named variables to each argument to create self-documenting code

```
$seq = make_sequence(50);

# create a random $len-mer
sub make_sequence {
  # create argument variable
  $len = $_[0];
  @bp = qw(a t g c);
  $seq = “”;
  for (1..$len) {
    $seq .= $bp[rand(@bp)];
  }
  return $seq;
}
```

BAD USAGE

```
$seq = make_sequence(50);

# create a random $_[0]-mer
sub make_sequence {
  @bp = qw(a t g c);
  $seq = “”;
  # access @_ directly
  for (1..$_[0]) {
    $seq .= $bp[rand(@bp)];
  }
  return $seq;
}
```

Challenge

what does square(5) return?

```
print square(5);

# there is a bug here
sub sum {
  my $num = @_;
  return $num**2;
}
```

Named Arguments

- Perl does not natively support named arguments
  - arguments passed as a list arrive in the same order and you need to remember the order when calling the function
  - recall that a hash is a 2n-element list – pass in a hash with keys as variable names

```
%hash = (len=>50,bp=>”atg”);
$seq = make_sequence(%hash);
$seq = make_sequence(len=>10, bp=>”at”);
$seq = make_sequence(bp=>”gcn”, len=>5);

# create a random n-mer from a specified vocabulary
sub make_sequence {
  # we are coercing an array to be stored as a hash
  # will break if @_ has odd number of elements
  %args = @_;
  @bp = split(“”,$args{bp});
  $seq = “”;
  for (1..$args{len}) {
    $seq .= $bp[rand(@bp)];
  }
  return $seq;
}
```

Checking Argument Integrity

- it’s very wise to check the integrity of arguments before using them
  - recall the difference between `if $x` and `if defined $x`

```
# create a random n-mer from a specified vocabulary
sub make_sequence {
  # we are coercing an array to be stored as a hash
  # will break if @_ has odd number of elements
  %args = @_;
  if(! length($args{bp})) {
    print “empty vocabulary string”;
    return undef;
  }
  if(! defined $args{len} || $args{len} < 0) {
    print “undefined or negative sequence length”;
    return undef
  }
  @bp = split(“”,$args{bp});
  $seq = “”;
  for (1..$args{len}) {
    $seq .= $bp[rand(@bp)];
  }
  return $seq;
}
```

Default Arguments

- if arguments fail checks, it is customary to assign default values

```
sub some_function {
  ($a,$b) = @_;
  # sets $a=10 if $a is false (i.e. 0 is considered unacceptable)
  $a ||= 10;
  # sets $b=10 if $b is not defined (i.e. 0 is considered acceptable)
  $b = 10 if ! defined $b;
  ...
}
```

- ||= operator is helpful here
  - `$a ||= 5` → `$a = $a || 5`

Returning Different Kinds of Variables

- subroutines may return any kind of variable
  - the caller must be aware of the behaviour of the subroutine

```
$x = sub1();
sub sub1 {
  ...
  return $x; # returns a scalar
}

@y = sub2();
sub sub2 {
  ...
  push @y, 10;
  ...
  return @y; # returns an array
}

%z = sub3();
sub sub3 {
  ...
  $z{red} = “apple”;
  ...
  return %z; # returns a hash
}
```

return Context

- this is a tricky point, but extremely important – *sit comfortably*
- recall that context is very important when cross-assigning variables
  - `$scalar = @array` has special meaning
- consider a function that returns N random numbers

```
# return N uniform random deviates
sub urds {
  my ($n) = @_;
  @urds = ();
  for (1..$n) {
    push @urds, rand;
  }
  return @urds;
}
```

return Context

- now look at how urds(3) behaves in these two situations

```
print urds(3);
0.329912258777767 0.549033692572266 0.577604257967323

print 1+urds(3);
4
```

- in the first case, print takes a list as its argument and therefore urds(3) is called in array context
- in the second case, + takes two scalars as arguments thus urds(3) is called in scalar context

return Context

- consider this function which returns a list of filtered base pairs
  - given $seq, return a list of base pairs in this string that are one of the characters in $testbp

```
# return base pairs from $seq that match $testbp
sub filter_seq {
  ($seq,$testbp) = @_;
  @passedseq = ();
  for $bp (split(“”,$seq)) {
    # pass $bp if it is matched by character chass [$testbp]
    # i.e. if it matches one of the characters in $testbp
    push @passedseq, $bp if /[$testbp]/;
  }
  return @passedseq;
}

print filter_seq(“aaatttgggccc”,”ag”);                # (a a a g g g)
$num_filtered = filter_seq(“aaatttgggccc”,”ag”);     # 6
# ($x) = @array – idiom for getting the first element out of the array
($num_filtered) = filter_seq(“aaatttgggccc”,”ag”);   # a
```

return Context

- do not assume how your function will be used
- if you mean to return a scalar value and there is possibility of it being evaluated in array context and returning a list
  - `return scalar`

```
# return base pairs from $seq that match $testbp
sub filter_seq {
  ($seq,$testbp) = @_;
  @passedseq = ();
  for $bp (split(“”,$seq)) {
    push @passedseq, $bp if /[$testbp]/;
  }
  return scalar @passedseq;
}

print filter_seq(“aaatttgggccc”,”ag”);               # 6
$num_filtered = filter_seq(“aaatttgggccc”,”ag”);     # 6
($num_filtered) = filter_seq(“aaatttgggccc”,”ag”);   # 6
```

Returning Failure

- you may wish to return failure to indicate that something has gone wrong
- in light of the previous slides, you should be hearing a warning klaxon
- how do you ensure failure in multiple contexts?
- we know enough not to `return 0`, so how about `return undef`

```
sub simulate_failure{
  return undef;
}

$x = simulate_failure();
print “failure scalar x” if ! defined $x;

@x = simulate_failure();
print “failure array x” if ! defined @x;

failure scalar x
```

- `oops!`

Returning Failure

- why did defined @x return true?

```
sub simulate_failure{
  return undef;
}

@x = simulate_failure();
# @x is now (undef), a list with a single undef element
# this list evaluates to true
```

- a bare return will always return strong failure (fails defined test) in the appropriate context

```
sub simulate_failure{
  return;
}

@x = simulate_failure();
# @x is now truly undefined, a list that fails defined check
```

1.0.8.1.6

Introduction to Perl
Session 6

- you now know
  - $_ and @_
  - subroutines
  - more about context
- next time
  - more on string manipulation
  - replacement and transliteration operators
  - global searches
  - contextual behaviour of =~