ANALYSIS: IMPORTANT

DNA on 10th — street art, wayfinding and font

BIOINFORMATICS + DATA VISUALIZATION

Learning Circos

BIOINFORMATICS AND GENOME ANALYSIS

FONDAZIONE EDMUND MACH, SAN MICHELE ALL'ADIGE, ITALY, 20 JUNE 2019

DOWNLOAD COURSE MATERIALS
v1.00 15 June 2019

A 1-day practical course in genomic data visualization with Circos. This material is part of the Bioinformatics and Genome Analysis course held at the Fondazione Edmund Mach in San Michele all'Adige, Italy.

QUICK LINKS

Course website | Circos documentation best practices getting started | Brewer palette swatches | Color resources | Points of View | Points of Significance

☰ DAY 1 SCRIPTS REFERENCES UPDATES

sessions / day.1

GENOMIC DATA VISUALIZATION WITH CIRCOS

Thursday 20 June 2019 — Day 1

09h00 – 10h30 | Lecture 1 — Introduction to Circos

11h00 – 12h30 | Lecture (practical) 2 — Visualizing gene distribution and size in Yeast—the histogram data track

14h00 – 15h30 | Lecture (practical) 3 — Conservation in Yeast—the link data track

16h00 – 18h00 | Lecture (practical) 4 — Drawing the human genome

18h15 – 19h30 | Lecture (practical) 5 — Afterhours—Perl refresher

18h15 – 19h30 | Lecture (practical) 6 — Afterhours—Visualizing an Ebola strain

---

CONCEPTS COVERED TODAY

Circos configuration, common Circos errors, Circos debugging, ideograms, selecting ideograms with regular expressions, input data format, creating Circos data files, data tracks (histograms, heat map, tiles, links), color definitions and using transparency, Brewer palettes, dynamic data formatting rules, downloading files from UCSC genome browser, essential command-line tools and basic scripting,

☰ LECTURE 1 LECTURE 2 LECTURE 3 LECTURE 4 LECTURE 5 LECTURE 6

sessions / day.1 / lecture.5

AFTERHOURS—PERL REFRESHER

sessions / day.1 / lecture.5 / README

These demonstrations are all in the **refresher** script in this directory, which is also listed below.

Run the script and follow along.

SCALARS, LISTS AND HASHES

The main three variable types you'll be interested in are scalars, lists and hashes.

```
my $x = 1                 # a scalar
my @x = (1,2,3)           # a list
my %x = (a=>1, b=>2, c=>3) # a hash
```

To pretty-print the data strcutures and understand what's inside, use the Data::Dumper

```
use Data::Dumper;

print(\@x);

# $VAR1 = [
#           1,
#           2,
#           3
#         ];

print(\%x);

# $VAR1 = {
#           'c' => 3,
#           'a' => 1,
#           'b' => 2
#         };
```

You'll often want to use references to lists and hashes. The reason for this is several-fold. First, passing references around is faster than the whole object, since in the latter case a copy of the object is made.

Second, compound data structures like lists of lists and lists of hashes (and even more deeper nesting) requires you to use references. Thus, you might as well use them from the top level of the data structure.

Third, if you want to change (add to, remove, modify, etc) a data structure in a function, you only need to pass it the reference and you don't have to return a new copy of the data structure (unless you want to). By passing a reference you can make changes in place. Unlike Python, which does many things in the background and you don't need to worry as much about references, Perl makes the distinction between variables and their references explicit.

```
my $xlistref = \@x;
my $xhashref = \%x;
```

The **refresher** script, listed below, hits some of the highlights that you should know.

To learn more, see learn.perl.org, Beginning Perl and Impatient Perl.

sessions / day.1 / lecture.5 / refresher

```
$\ = "\n"; # default end of line delimiter
$, = " ";  # default field delimiter

my $x = 1;
my @x = (1,2,3);
my %x = (a=>1,b=>2,c=>3);

# Note that $x @x and %x are all DIFFERENT variables, even
# though they have the same name. There is the scalar x, list x and hash x.
# For now, consider them to be independent and unrelated.
#
# Avoid doing things like
#
# @xlist = (1,2,3)
#
# since @xlist is obviously a list.
#
# It's generally bad practise to add the variable type to its name. However,
# I do this in this script for clarity. For example, the variable $xlistref
# would probably just be called $things or $fruits.
#
# Treat the words "list" and "array" as synonymous. Technically, an array
# is the data type and the list is the thing that it contains, but this
# is a minor distinction. For compound lists (e.g. lists of lists) you would
# say "two-dimensional array" and not "two-dimensional list".

print ("if we print a list we get",@x);
print ("if we print a hash we get",%x);

print ("list via Data::Dumper");
print Dumper(\@x);
print ("hash via Data::Dumper");
print Dumper(\%x);

my $xlistref = \@x;
my $xhashref = \%x;

print ("xlistref",$xlistref,"is reference to",ref $xlistref);
print ("xlistref",$xhashref," is reference to",ref $xhashref);

# Dereference the references back to their original types

my @y = @$xlistref;
```

```
my %y = %$xhashref;

# looping over a list

for my $item (@x) {
  print ("looping over list",$item);
}

# when you get comfortable with the idea that $x and @x
# are different variables, you can do this

for my $x (@x) {
  print ("looping again over list",$x);
}

for my $i (0..@x-1) {
  print ("looping over list index",$i,"item",$x[$i]);
}

# looping over a reference of the list - requires that
# we dereference the list by prepending the list sigil @
# to the variable.
#
# to access a list item via reference, you use ->

for my $i (0..@$xlistref-1) {
  print ("looping over list ref index",$i,"item",$xlistref->[$i]);
}

# looping over keys in a hash
for my $key (keys %x) {
  print ("key",$key,"value",$x{$key});
}

# looping over keys in a hash reference
#
# to access a hash item via reference you use ->
for my $key (keys %$xhashref) {
  print ("key",$key,"value",$xhashref->{$key});
}

# variables can be evaluated in a context to force
# an "interpretation" of the variable

# when a list is evaluated in scalar context (i.e. it is
# being assigned to a variable that is explicitly a scalar)
# you get the number of items in the list.

my $scalar = @x;
print("list as scalar",$scalar);

# Note that many contexts are list context such as
#
# print @x
# for (@x)
#

# Since perl 0-indexes lists, the last object is at index

my $lastidx = @x-1;
print("last index",$lastidx);

# The loop below cycles over the indexes in @x

for my $i (0..@x-1) {
  print("index",$i,"value",$x[$i]);
}

# You can also access the last index via the special variable $#x
# so you can write the loop like this

for my $i (0..$#x) {}

# Personally, I do not like this notation and always write @x-1. This
# special variable is very very Perlish (Perl has lots of internal
# variables like this and it's up to you how many of them you want to use).
#
# http://perldoc.perl.org/perlvar.html

# For example, the current iterator in a for loop is accessed by $_

for (@x) {
  print("current item",$_);
}

# Evaluating a scalar in list context
```

```
$x = "hello";
my @list = $x;

# simply gives you a single element list ("hello").

# To quickly define lists, use the word quotation operator qw()

@list = qw(some words in a list);

# This is much faster than

@list = ("some","words","in","a","short","list");

# If you evaluate a list in a hash context, Perl will expect an
# even number of elements and assign them as key/value pairs. Notice
# that Perl will interpolate variables inside a string defined
# with double quotes.

my %wordhash = @list;
for my $key (keys %wordhash) {
  print "$key -> $wordhash{$key}";
}

# One of the reasons why Perl was very popular in its day was its
# support for regular expressions
#
# https://perldoc.perl.org/perlre.html

my $str = "abccddfffhiiijj";
if($str =~ /c/) {
  print($str,"matches");
}

# For example to match any pair of identical characters you would
# use (.)\1 where . is the character class for any character, the
# (.) is a capture bracket which stores the match in an internal
# variable accessible immediately by \1.
#
# Then during replacement this internal variable is available as $1. The
# notation is different (e.g. vs \1) because we're no longer in the middle
# of doing the match. The match is done and we're now replacing strings.

$str =~ s/(.)\1/[2 x $1]/g;
print("replaced",$str);

# Regular expressions are a big topic, but you only need to know a few
# important things to get started. You can test-drive your regular expressions here
#
# https://regex101.com/

# You'll often be splitting strings into fields.

$str = "10 20    30 55   102 28";

# Using " " tells split to treat any amount of whitespace as a delimiter

my @tok = split(" ",$str);
print("split line into",@tok);
print("second field is",$tok[1]); # remember lists are 0-indexed

# You can now select items that you want

for my $tok (@tok) {
  if($tok eq 10) { # string comparison
    # ...
  }
  if($tok =~ /0/) {
    print("item",$tok,"has a 0");
  }
}

# Alternatively, a Perlish thing to do is to apply grep() to the list.
# Trep takes a condition as the first argument, which can be the
# regular expression

for my $tok (grep(/0/, @tok)) {
  print("item",$tok,"has a 0 via grep");
}

# Strictly speaking grep(EXPRESSION,LIST) passes each value from LIST
# to the EXPRESSION via $_ and the expression is evaluated. If it
# evaluates to true then grep returns the list item, otherwise it does not.

for my $tok (grep($_ =~ /0/, @tok)) {}
```

```
# Grep is a *great* way to filter a list.

# To apply code to each list item, use map()

my @newtok = map { 2*$_ } @tok;
print("remapped list",@newtok);

# The code in { } can be anything and the new list is composed of the
# values returned by the code, as evaluated on each list element via $_

# You can combine map { } and grep(). For example, here we
# multiply each element that contains a 0 by 2. The list is not
# modified - both grep and map return copies of the elements.

print("map{} and grep()",map { 2*$_ } grep(/0/, @tok));

# To pull out all numbers made out of digits from a string

my $input = "aa1bb22ccc333dd444";
while( $input =~ /(\d+)/g ) {
  print "Found $1 in $input";
}

# There is a lot going on here. The regular expression operator =~ applies
# the regular expression (\d+) to $input. Because the regular expression has capture
# brackets and we're using /g and this is evaluated in a while loop, the operator
# will continue to return as many matches as it can find. The substring matched will be stored in the
# special variable $1, which you can think of as the result of the first capture bracket.

# Below I add a second set of capture brackets to the regular expression, which grabs
# the letters immediately following the digit. Here the + means "one or more".

while( $input =~ /(\d+)([a-z]+)/g ) {
  print "Found digit/letter combo $1/$2 in $input";
}

# A final useful regular expression character are the string anchors ^, which
# matches the start of a string, and $, which matches the end. Thus

if($input =~ /\d$/) { }

# will match if $input ends in a digit and

if($input =~ /^\d/) { }

# will match if it starts with a digit. To check that the string is made up exclusively
# of digits you would do this

if($input =~ /^\d$/) { }
```