A very short introduction to GAP

- GAP helps to compute small examples or search for counterexamples. It provides
  - good possibilities to compute with groups, many useful functions are already implemented in GAP,
  - libraries of groups, character tables etc.
  - a possibility to use and combine these functions.
- GAP is included in many package managers or can be downloaded from

  www.gap-system.org

  On this website, under “Documentation” → “Manuals” you can also find in `pdf` and `html`-format
  - a “Tutorial” (ca. 80 pages, a short introduction including “A First Session with GAP”)
  - the “Reference Manual” (ca. 1400 pages, a detailed description of most available functions including examples, can be searched through.)
- GAP can be exited by typing `quit;`
- Each GAP-comand is finished by a semicolon. If you forget this, you have to add it in the next line. Two semicolons prevent printing of the output, which is particularly useful for long outputs.
- After a “wrong input” GAP will enter a “break loop” and print `brk>` (or, if you are already in a break loop it will print, `brk_02>`, `brk_03>` and so forth). You can leave the most inner break loop by typing `quit;`
- You can log your session, e.g. into file.log in the current directory by typing `LogTo(”file.log”);`. To end the logging type `LogTo();`
- Taping “↑” shows you the last entered command.
- Taping the tabulator completes a command, if it can be completed uniquely. Double taping the tabulator shows all possible completions.
- Typing a question mark before a command shows the help for this command. E.g. `?Comm;` shows the help for the command `Comm`, which commutes a commutator.
  You can change the way GAP displays help using the `SetHelpViewer` command.
  E.g. `SetHelpViewer("firefox");`
- Variables are set using `:=`
- GAP includes many packages which are “Add-Ons” of the basic functions of GAP and provide functionality for special types of calculations. A package which has not been loaded yet in a session can be activated by typing `LoadPackage("name");` where `name` must be replaced by the name of the package. E.g. `LoadPackage("wedderga");` will be very helpful this week.

- Computing with permutations:
  - Permutations are entered and printed in the usual notation using disjoint cycles: `f := (1,4)(2,5,3);` and `g := (2,3);` gives the permutations

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 2 & 1 & 3 \end{pmatrix} \qquad \text{and} \qquad g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 2 & 4 & 5 \end{pmatrix}$$

  - Permutations are multiplied using `*`. Here `f*g;` means that f is applied first and g second. So here we get `(1,4)(2,5)`
  - `()` denotes the identity in a permutation group.
  - `g^-1;` computes the inverse of g. By `g^f;` we get the conjugate $f^{-1}gf$ of g by f. Here we get `(2,5)`
  - Permutation groups can be defined as `G := SymmetricGroup(5);` or by entering generating elements, e.g. `H := Group((2,3,4,5,6), (2,3));` or a list of generating elements `K := Group([(1,2,3), (3,4,5)]);`. Here H generates a group isomorphic to S_5, but acting on the set $\{2,3,4,5,6\}$. As by default in `GAP` the group `SymmetricGroup(5)` acts on the first five integers, the input `G = H;` returns `false`
  - The command `IsomorphismGroups(G, H);` searches for an isomorphism between G and H and returns such an isomorphism if it finds one. Otherwise it returns `fail`. In the example above we will hence get an isomorphism between G and H.
  - The order of a group element can be found using the command `Order`.
    E.g. `Order((1,2)(3,4,5));` returns `6`.

- Computing with matrices.
  - Matrices are entered in `GAP` using lists of lists. E.g. `A := [[2,1],[-1,3]]` will write the matrix $\begin{pmatrix} 2 & 1 \\ -1 & 3 \end{pmatrix}$ into the variable `A`.
  - The usual arithmetic operations `+, -, *, /` are all available for matrices. Also `A^-1` will return the inverse of an invertible matrix `A`.

- On conjugacy classes. The example A_5
  - The command `CCG := ConjugacyClasses(AlternatingGroup(5));` sets the variable `CCG` to a list containing the conjugacy classes of A_5. Here it gives

    ```
    [ ()^G, (1,2)(3,4)^G, (1,2,3)^G, (1,2,3,4,5)^G, (1,2,3,5,4)^G ]
    ```

  - The i-th entry of the list `CCG` can be accessed by `CCG[i];`.
  - A representative of an element in `CCG` can found using the command `Representative`. E.g. `Representative(CCG[2]);` gives `(1,2)(3,4)`.
  - The command `List` can be used to apply the same command to all elements of a list. E.g. `List(CCG, x -> Size(x));` will return the length of all conjugacy classes in A_5: `[1, 15, 20, 12, 12]`.

- A short function: Are there square roots?
  - Using `GAP` we want to check, if a given finite group G contains a square root for every element, i.e. if all elements in G are squares. We write a function `SquareIsOnto(G)`, which takes a finite group G as input and returns `true` if the map $g \mapsto g^2$ is surjective. Otherwise it returns `false`.

- Such a function can be saved in a separate file which can then be copied into the GAP-session or read using the command `Read`. Here our function could look like this:

```
SquareIsOnto := function(G)
local squares, x;   # Declare local variables
squares := [ ];      # List which will contain all squares
for x in G do       # go through all elements in G
    Add(squares, x^2); # Write
od;
squares := DuplicateFreeList(squares);
                    # Delete elements appearing multiple times
if Size(squares) = Order(G) then
    return true;
else
    return false;
fi;
end;
```

  In a shorter way the same can be achieved by:

```
SquareIsOnto := function(G)
return Size(DuplicateFreeList(List(G, g -> g^2))) = Order(G);
end;
```

- Some other useful GAP-functions:
  - Special groups:
    * `CyclicGroup(n)` generates a cyclic group of order n.
    * `AlternatingGroup(n)` generates an alternating group of degree n.
    * `DihedralGroup(n)` generates the dihedral group D_n of order n.
    * `GL(n,q)`, the general linear group of $n \times n$-matrices over a filed with q elements.
    * `DirectProduct(G,H)` will return the direct product of the two groups `G` and `H`.
    * `G/N` the quotient group of `G` by the normal subgroup `N`.
  - Some subgroups of a group:
    * `AllSubgroups(G)` returns a list of all subgroups of G.
    * `NormalSubgroups(G)` returns a list of all normal subgroups of G.
    * `SylowSubgroup(G,p)` returns a Sylow p-subgroup of G.
    * `Centralizer(G,U)` returns the centralizer of U in G.
    * `Normalizer(G,U)` returns the normalizer of U in G.
    * `Centre(G)` or `Center(G)` returns the center of G.
    * `DerivedSubgroup(G)` returns the derived group G'.
  - Testing properties:
    * `IsSolvable(G)` checks if G is solvable.
    * `IsPGroup(G)` checks if G is a p-group.
    * `IsCyclic(G)` checks if G is cyclic.
    * `IsAbelian(G)` checks if G is abelian.

    * `IsSubgroup(G,U)` checks if U is a group and a subset of the group G.
    * `IsNormal(G,N)` checks if N is a normal subgroup of G.
- Integers:
    * `IsPrimeInt(n)` tests if n is a prime.
    * `IsPrimePowerInt(n)` tests if n is a prime power.
    * `FactorsInt(n)` returns the prime factors of n with their multiplicity.
- Lists:
    * `Filtered(L, x -> f(x))` returns those of the entries in the list `L` for which the function `f` returns `true`. E.g. `Filtered(AllGroups(27), x -> IsAbelian(x))` returns all abelian groups of order 27.
    * `Positions(L, x)` returns the positions in which the list `L` contains the element `x`. If `x` is not contained in `L` it returns `[ ]`.
- More:
    * `StructureDescription(G)` gives a description of G in terms of "known" groups, if it can find such a description.
    * `Order(G), Order(g)` returns the order of a group or an element.
    * `AllGroups(n)` returns a list of all groups of order n up to isomorphism (only for small n).
    * A finite field of order q is denoted by `GF(q)` and multiplicatively generated by `Z(q)`.