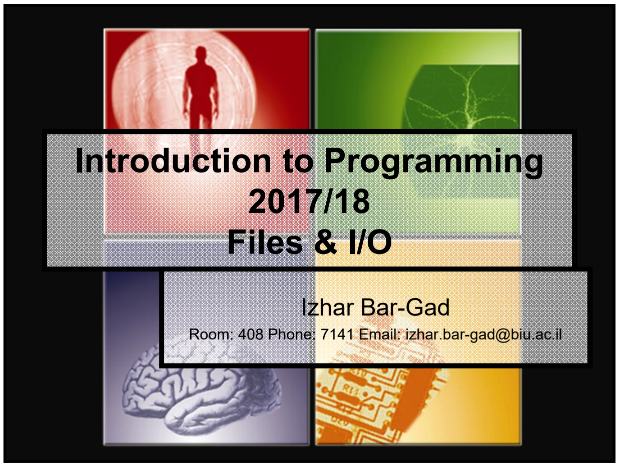
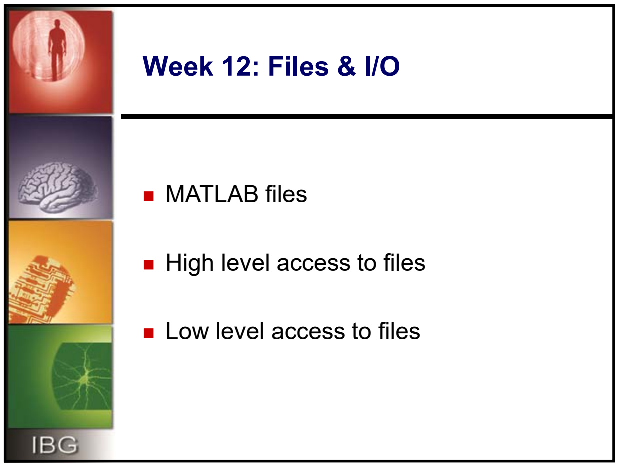


Introduction to Programming 2017/18 Files & I/O

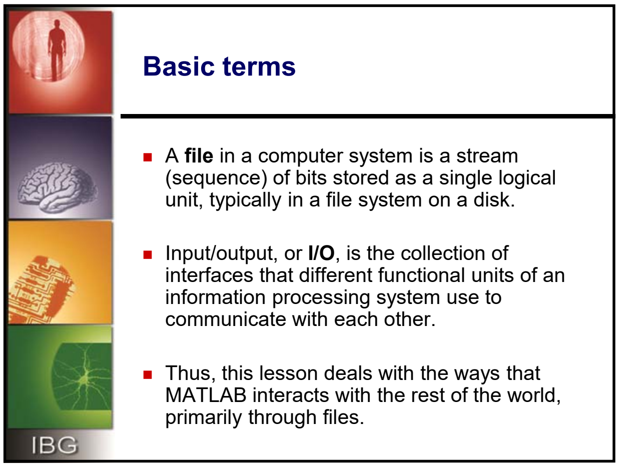
Izhar Bar-Gad

Room: 408 Phone: 7141 Email: izhar.bar-gad@biu.ac.il



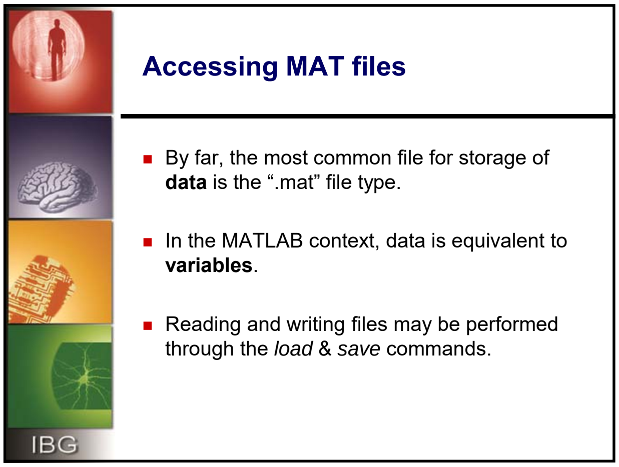
Week 12: Files & I/O

- MATLAB files
- High level access to files
- Low level access to files



Basic terms

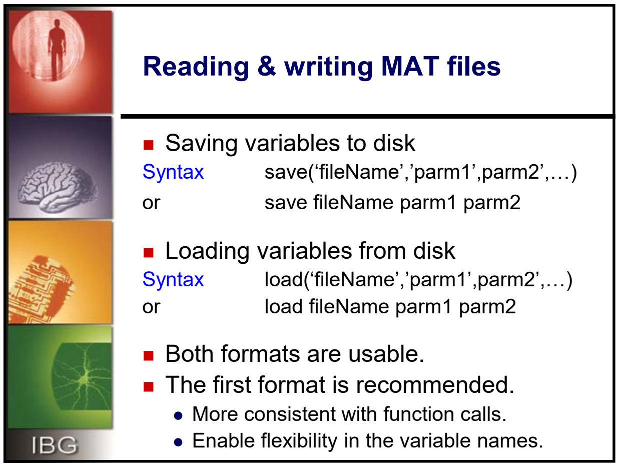
- A **file** in a computer system is a stream (sequence) of bits stored as a single logical unit, typically in a file system on a disk.
- Input/output, or **I/O**, is the collection of interfaces that different functional units of an information processing system use to communicate with each other.
- Thus, this lesson deals with the ways that MATLAB interacts with the rest of the world, primarily through files.



Accessing MAT files

- By far, the most common file for storage of **data** is the “.mat” file type.
- In the MATLAB context, data is equivalent to **variables**.
- Reading and writing files may be performed through the *load* & *save* commands.

IBG



Reading & writing MAT files

- Saving variables to disk

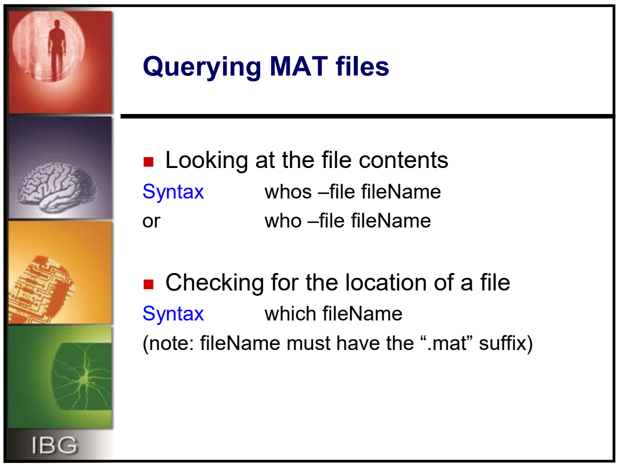
Syntax save(‘fileName’,’parm1’,parm2’,…)
or save fileName parm1 parm2

- Loading variables from disk

Syntax load(‘fileName’,’parm1’,parm2’,…)
or load fileName parm1 parm2

- Both formats are usable.
- The first format is recommended.
  - More consistent with function calls.
  - Enable flexibility in the variable names.

IBG



Querying MAT files

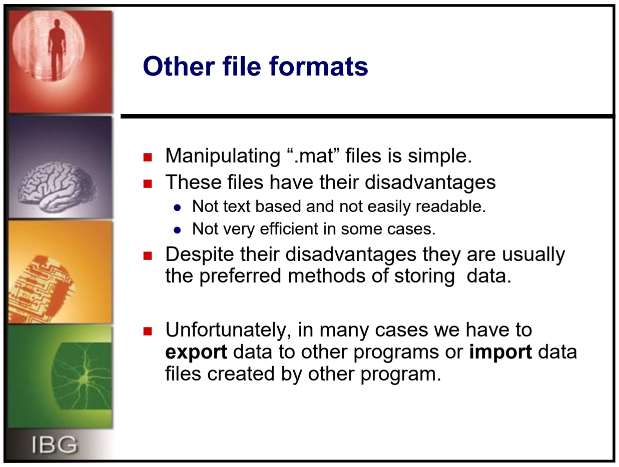
- Looking at the file contents

Syntax whos –file fileName
or who –file fileName

- Checking for the location of a file

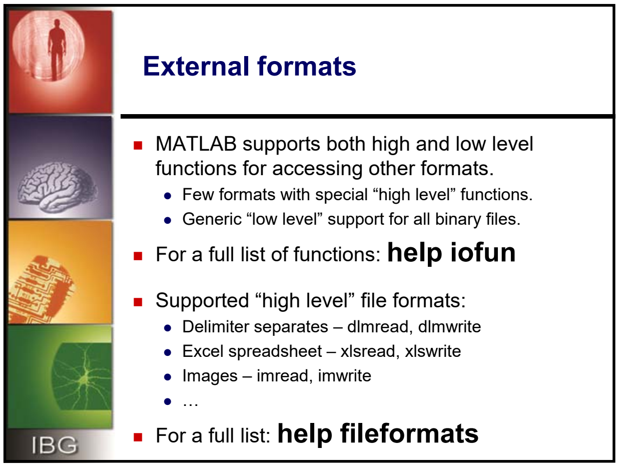
Syntax which fileName
(note: fileName must have the “.mat” suffix)

IBG



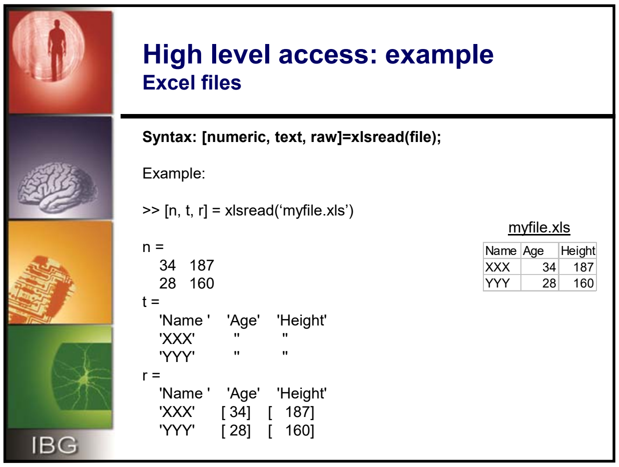
Other file formats

- Manipulating “.mat” files is simple.
- These files have their disadvantages
  - Not text based and not easily readable.
  - Not very efficient in some cases.
- Despite their disadvantages they are usually the preferred methods of storing data.
- Unfortunately, in many cases we have to **export** data to other programs or **import** data files created by other program.



External formats

- MATLAB supports both high and low level functions for accessing other formats.
  - Few formats with special “high level” functions.
  - Generic “low level” support for all binary files.
- For a full list of functions: **help iofun**
- Supported “high level” file formats:
  - Delimiter separates – dlmread, dlmwrite
  - Excel spreadsheet – xlsread, xlswrite
  - Images – imread, imwrite
  - …
- For a full list: **help fileformats**



High level access: example
Excel files

Syntax: [numeric, text, raw]=xlsread(file);

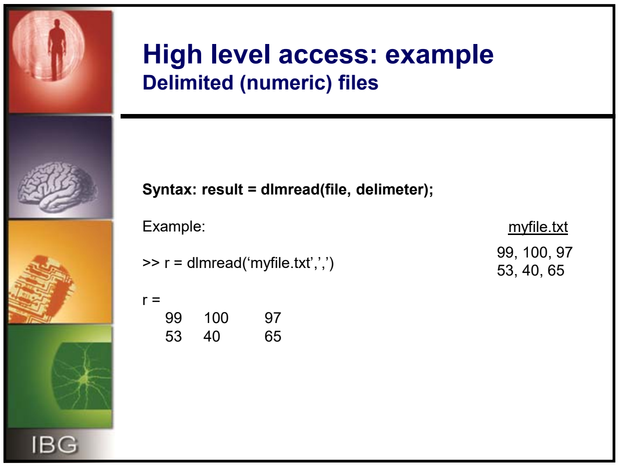
Example:

```
>> [n, t, r] = xlsread(‘myfile.xls’)

n =
   34   187
   28   160
t =
   'Name '   'Age'   'Height'
   'XXX'       ''         ''
   'YYY'       ''         ''
r =
   'Name '   'Age'   'Height'
   'XXX'     [ 34]   [   187]
   'YYY'     [ 28]   [   160]
```

myfile.xls

Name	Age	Height
XXX	34	187
YYY	28	160



High level access: example

Delimited (numeric) files

Syntax: result = dlmread(file, delimeter);

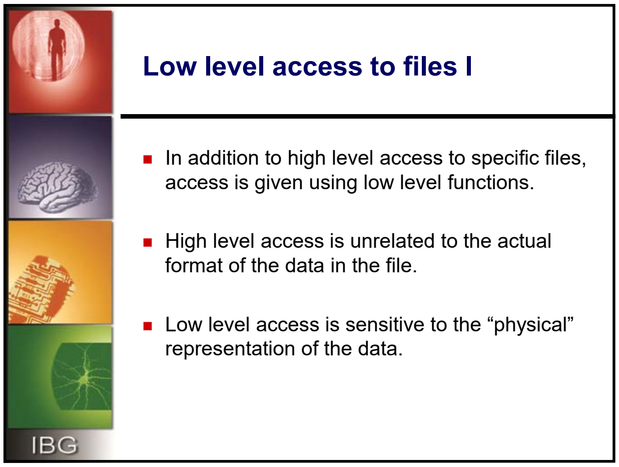
Example:

```
>> r = dlmread('myfile.txt',',')

r =
    99    100       97
    53    40        65
```

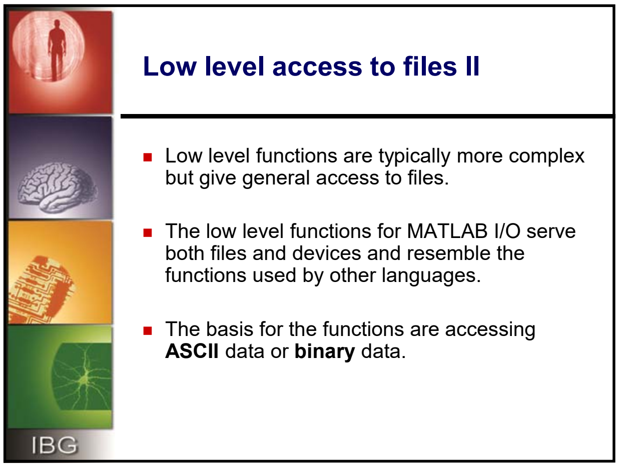
myfile.txt

```
99, 100, 97
53, 40, 65
```



Low level access to files I

- In addition to high level access to specific files, access is given using low level functions.
- High level access is unrelated to the actual format of the data in the file.
- Low level access is sensitive to the “physical” representation of the data.



Low level access to files II

- Low level functions are typically more complex but give general access to files.
- The low level functions for MATLAB I/O serve both files and devices and resemble the functions used by other languages.
- The basis for the functions are accessing **ASCII** data or **binary** data.

Opening a file

- The file must first be opened with the required attributes.

Syntax: fid = fopen(fileName, permissions)

- Success: a positive identifier (typically >2).
  - 1 is standard output & 2 is standard error
- Failure: a negative (-1) identifier.

- Permissions – mode of access to the file

r – read
w – write (& create)
r+ – read & write (no creation)
a – append
…

Closing a file

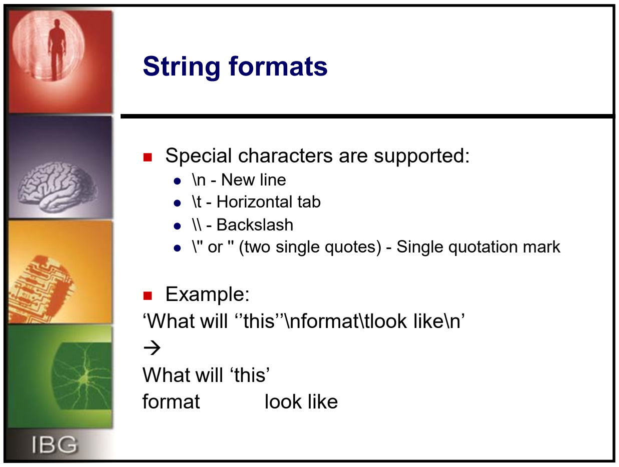
- The file must be closed. Otherwise it will stay locked and will be problematic to access.

Syntax: status = fclose(fid)

- Status is 0 upon success and -1 upon failure.

Handling text files

- Text files are written using 1 byte per character.
- Most characters are printable with a few exceptions for control characters.
- Thus, text files are readable to humans.
- The **encoding** of the character is the transformation between the binary value of the byte to the displayed character.
- **ASCII** (American Standard Code for Information Interchange) – the most common character encoding for describing text on a computer



String formats

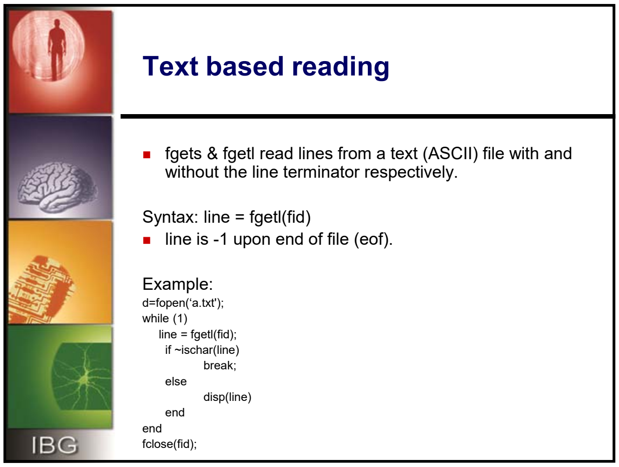
- Special characters are supported:
  - \n - New line
  - \t - Horizontal tab
  - \\ - Backslash
  - \" or '' (two single quotes) - Single quotation mark
- Example:

‘What will ‘’this’’\nformat\tlook like\n’

→

What will ‘this’

format   look like



Text based reading

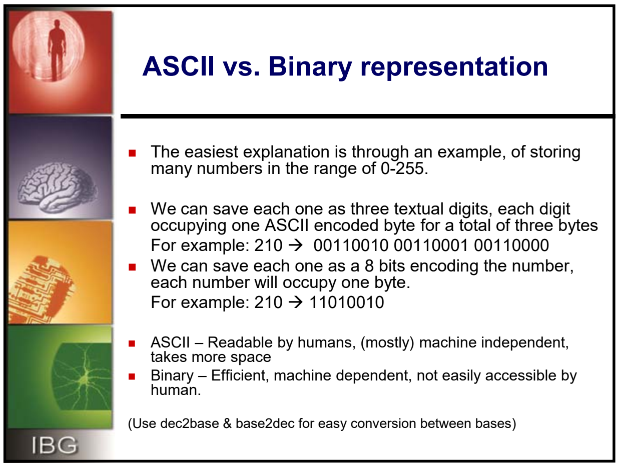
- fgets & fgetl read lines from a text (ASCII) file with and without the line terminator respectively.

Syntax: line = fgetl(fid)

- line is -1 upon end of file (eof).

Example:

```
d=fopen(‘a.txt');
while (1)
    line = fgetl(fid);
      if ~ischar(line)
                break;
      else
                disp(line)
      end
end
fclose(fid);
```



ASCII vs. Binary representation

- The easiest explanation is through an example, of storing many numbers in the range of 0-255.
- We can save each one as three textual digits, each digit occupying one ASCII encoded byte for a total of three bytes
  For example: 210 → 00110010 00110001 00110000
- We can save each one as a 8 bits encoding the number, each number will occupy one byte.
  For example: 210 → 11010010
- ASCII – Readable by humans, (mostly) machine independent, takes more space
- Binary – Efficient, machine dependent, not easily accessible by human.

(Use dec2base & base2dec for easy conversion between bases)

Binary data

- Unlike text based access, the representation of the data is crucial.

- Integer vs. floating point
- Signed vs. unsigned
- Number of bytes used per variable

IBG

Binary representation of data

- The basic binary representations are:
  - Character
    - Elements of strings.
    - Uses 1 byte per element.
  - Integer – whole numbers (-min, .., -1, 0, 1, …, max)
    - Signed or Unsigned
    - Uses 1-8 bytes per element.
    - Uses 2-16 for complex elements.
  - Floating point – real numbers
    - Single – Uses 4 bytes per element (8 for complex)
    - Double – Uses 8 bytes per element (16 for complex)

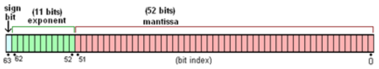
IBG

Binary data example

- The range of a signed 1 byte integer is
  [-128,127]
- The range of an unsigned 2 byte integer is
  [0, 65535]

- Double precision floating point

Max: 1.7976931348623157 x 10^{308}



$(-1)^{\text{Sign}} * 2^{(\text{Exponent - ExponentBias})} *$ 1.Mantissa

IBG

Binary reading and writing

- fread - Reading binary info

Syntax: outVar = fread(fid, numElems, precision)

- fwrite – Writing binary info

Syntax: count = fwrite(fid, outVar, precision)

- Reasons for using
  - Saves space in file.
  - Unfortunately many files you get will look like this.
- Reasons for not using
  - Unreadable to the human eye.
  - Extremely hard to debug.

IBG

Navigating within a file

- fseek – move within the file

Syntax: fseek(fid, offset, origin)

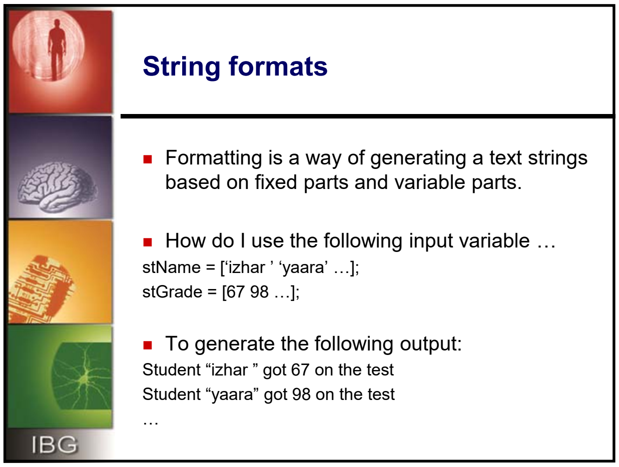
  - offset
    - negative → move backwards
    - positive → move forwards
  - origin
    - 'bof' or -1 → beginning of file
    - 'cof' or 0 → current position in file
    - 'eof' or 1 → End of file
  - status
    - 0 is success & -1 is failure
- ftell – find position within the file

Syntax: position = ftell(fid)

IBG

Additional material

IBG



String formats

- Formatting is a way of generating a text strings based on fixed parts and variable parts.
- How do I use the following input variable …

stName = ['izhar ' 'yaara' …];

stGrade = [67 98 …];

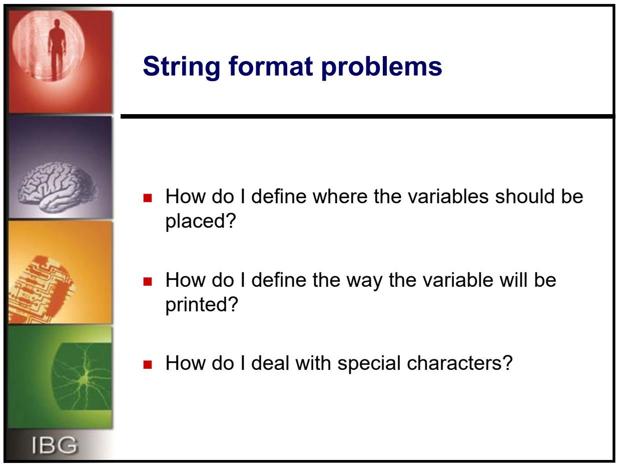
- To generate the following output:

Student "izhar " got 67 on the test

Student "yaara" got 98 on the test

…

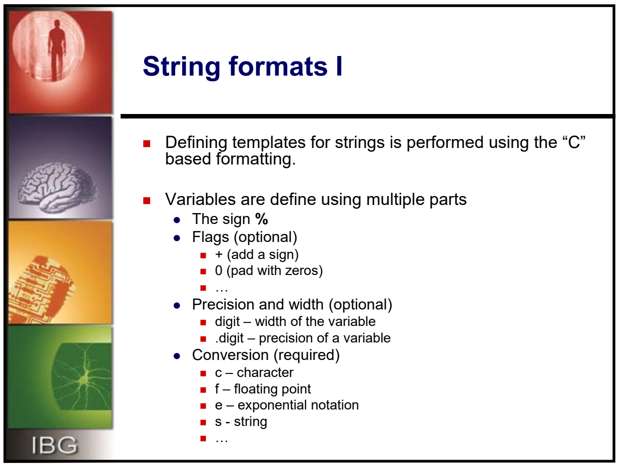
IBG



String format problems

- How do I define where the variables should be placed?
- How do I define the way the variable will be printed?
- How do I deal with special characters?

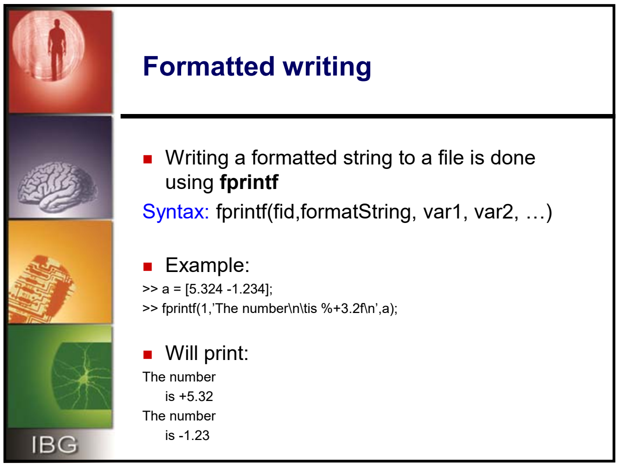
IBG



String formats I

- Defining templates for strings is performed using the "C" based formatting.
- Variables are define using multiple parts
  - The sign **%**
  - Flags (optional)
    - + (add a sign)
    - 0 (pad with zeros)
    - …
  - Precision and width (optional)
    - digit – width of the variable
    - .digit – precision of a variable
  - Conversion (required)
    - c – character
    - f – floating point
    - e – exponential notation
    - s - string
    - …

IBG



Formatted writing

- Writing a formatted string to a file is done using **fprintf**

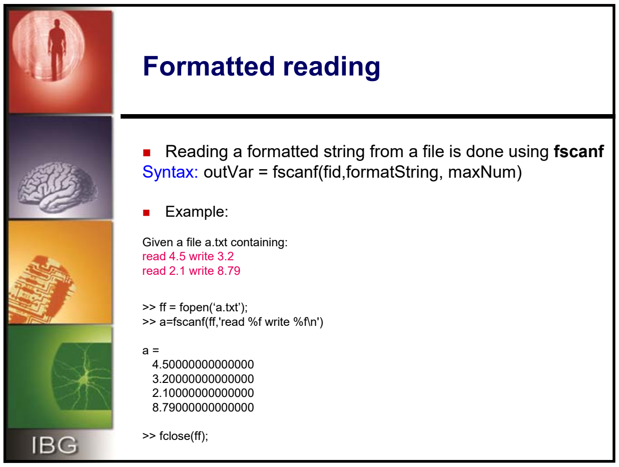
Syntax: fprintf(fid,formatString, var1, var2, …)

- Example:

```
>> a = [5.324 -1.234];
>> fprintf(1,’The number\n\tis %+3.2f\n’,a);
```

- Will print:

```
The number
     is +5.32
The number
     is -1.23
```



Formatted reading

- Reading a formatted string from a file is done using **fscanf**

Syntax: outVar = fscanf(fid,formatString, maxNum)

- Example:

Given a file a.txt containing:

```
read 4.5 write 3.2
read 2.1 write 8.79
```

```
>> ff = fopen(‘a.txt’);
>> a=fscanf(ff,'read %f write %f\n')

a =
   4.50000000000000
   3.20000000000000
   2.10000000000000
   8.79000000000000

>> fclose(ff);
```