**Week 1: C & Terminal Commands**

**Commands:**

* Cd “directory” – access directory. “.” = current, “..” = parent, “” = base
* Ls – lists files & folders
* pwd – returns current directory
* Mkdir “name” – make new folder
* rm “name[.type]” – remove file. -f to force. -r to do recursively (all children)
* mv “source” “destination” – move or rename
* cp “source” “destination” – copy file. -r for recursive (all children)
* code “name.type” – open or create file
* make “name” – compile program

**TODO:**

* chmod
* rmdir
* sudo
* ln
* man
* clear
* touch
* diff
* telnet

**Week 2: Data Types:**

**Data Types:**

**Integers:**Char – ascii character. Has int value & char value. Size 1  
int – integer. Size 4  
short – short integer  
long – long integer. Size 8  
**Qualifiers:**unsigned – positive only. for integers  
signed – positive & negative. For chars only.  
**Decimals:**float – 6 decimals. Size 4  
double – 15 decimals. Size 8  
long double – 19 decimals.

**Conversion:**

(type) value – changes data type of value into the type if possible. Truncates decimals if (int) float.

**Week 3: Algorithms – typedef, searches, sorts, recursive functions**

**Functions:**

**<string.h>**

* Strlen(string) – returns length of string
* strcmp(string, string) – compares 2 strings (0 = same)

**<ctype.h>**

* isupper(char) – returns true if upper
* islower(char) – returns true if lower
* isdigit(char) – returns true if digit
* tolower(char) – returns lower character OR (no change if already lower or symbol)
* toupper(char) – returns upper character OR (no change if already upper or symbol)

**<math.h>**

* ceil(float) – returns float rounded up value
* floor(float) – returns float rounded down value
* pow(float x , float y) – returns x^y
* sqrt(float x) – returns square root of x
* log(float x) OR log10(float x) – returns log(e) OR (10) of x

Echo $? = get return value of main function

**Typedef:**Syntax: typedef “oldname” “newname”  
Can define struct in the middle of typedef (struct = oldname)typedef struct [name of struct]  
{  
 basetype name1  
 basetype name2  
}  
newname  
varname.name1/2

**Searches:**Linear search – 1 🡪 n  
binary search – 1🡪log2(n)  
  
**Sorts:**  
Selection – n^2🡪n^2 (theta)  
bubble – n🡪n^2  
merge – nlog2(n)🡪nlog2(n)

**Recursive function normal structure:**base case {return}  
recursive case {f(x)}  
return

**Week 4: Memory – pointers, addresses, strings, images & pixels**

**Data type/constant:**

**Types:**uint8\_t – 8bit unsigned integer  
uint16\_t – 16bit unsigned integer  
**Constants**  
FILE – pointer to a file. Use via fopen()   
EOF – end of file character (constant)  
NULL – null value (returned by functions on error, cannot be accessed)  
NUL OR \0 – end of string

**Functions:**

**<string.h>**

* strcpy(char \*s, char \*t) – copy string from t to s

**<stdlib.h>**

* malloc(int x) – returns address (assign to pointer) of first free memory of x length
* calloc(qty, size) – returns address of memory of qty \* size length and initialise values to 0
* realloc(\*ptr, new size) – returns address of new memory of new size. Make sure to assign this to \*temp, as assigning this to existing ptr will lead to memory leak (the memory the ptr was previously referencing)
* free(pointer) – frees chunk of memory at address. Use every time you use malloc

**<stdio.h>**

* scanf(“%datatype”, &x) – scans keyboard for input, stores as specified datatype at address of x
* fopen(“name.type”, “operation”) – opens file & returns file pointer to it. CHECK FOR NULL.
  + Operations: r (read), w(write), a (append - write from end of file)
* fclose(FILE\*) – closes file
* fgetc(FILE\*) – returns next character from file. Must be “r” operation
* fgets() – gets string from file
* fputc(char, FILE\*) – writes/appends specified character to file. Must be “a” or “w” operation
* fputs() – writes string to file
* fprintf(FILE\*, “string%type”, variable) – writes formatted string to file
* fread(\*buffer, size, qty, \*FILE) – read qty number of size \* byte sized values to address pointed by buffer from file pointed by file pointer. buffer == array(stack)/malloc (heap)
* fwrite(\*buffer, size, qty, \*FILE) – write qty number of size \* byte sized values from buffer to file pointed by file pointer. buffer == array(stack)/malloc (heap)
* fseek() – rewind or fastfoward through file
* ftell() – returns byte position you are at within file
* feof() – returns bool whether you’ve read to the end of the file
* ferror() – returns bool whether error has occurred working in file

**Operators:**

**Sizeof():**  
sizeof(expression) – returns number of bytes of given data type. Useful to allocate sufficient memory with malloc()  
sizeof(array)/sizeof(array[0]) – calculate number of elements in arrays

**Pointer:**  
\*(DATATYPE): pointer to the address of a variable. Type of pointer = data located at address. Name of array = pointer to first address.  
\*(Dereference Operator): goes to the address of a specified pointer  
&: the address of a variable

x = address of the variable assigned to \*x  
\*x = value at the address stored in x  
\*(x+1) = value at address of x + (1 \* memory size of type of x). ie – value stored at address stored at next position of array irrespective of memory sizes (eg, 1 bit vs 4 bit)  
&x = address of variable x (which is the address of variable)

**Strings:**

char \*x: string of name x, which points to memory address of first char. x is created as an array of chars ending with NULL OR \0 after the end of the string.  
x = address of first char. Strings are special (you do not need to &x to reference the address)  
\*x OR \*x[0] = value at address of first char of string/array x  
\*x[n] OR \*(x+n) = value at the address of (n-1) char (as array is consecutive until NUL OR \0)  
&x = address of variable x (which is the address of first char)  
printf(“&s”, x) = prints char stored at the address of the value of (x, then x+1, … until value of x+n = NULL)  
“…” = string literal – ie cannot be changed. Comparisons compare memory addresses so you have to iteraterate each character OR use strcmp() in ctype.h

**Difference between char s\* and char s[]**

Both are used to create strings.   
\*s - pointer to the first char of string, which is stored separately from the pointer. Must malloc memory. Both s[n] and \*(s+n) can be used.  
s[n] – location named s of n size is allocated. When passed through functions, it is passed as a pointer. Both s[n] and \*(s+n) can be used.

**Manipulating pointers:**

char \*x = “string” 🡪 x (pointer) gets address of “string”  
char \*t = x 🡪 t (pointer) gets the value of x (the address of “string”)  
t[n] = x[m] 🡪 copies (m+1)th char of x (string) to (n+1)th char of t (string)  
char \*\*t = &x 🡪 t (pointer pointer) gets address of x (pointer)  
int x = number 🡪 x assigned value of number  
int \*y = number 🡪 number allocated at address that y is pointing to (must assign the pointee of y first)  
int \*y = &x 🡪 y (pointer) gets address of x (integer)  
int \*z = y 🡪 z (pointer) gets value of y (pointer)

**NULL:**

Pointer to first memory of computer that nothing should be assigned to.  
Some functions (eg, malloc, strcpy) return NULL when there is an error (eg, insufficient memory).  
Add breakpoints to break program if functions return NULL

**Reference vs value:**

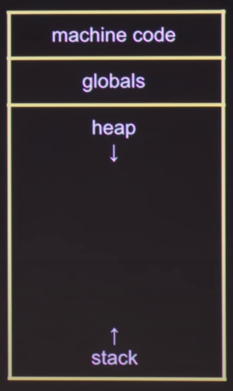
When passing arguments into functions, the value is given. If the value is changed inside the function, it does not change the original arguments that were passed in (as it did not affect the values assigned to the specific addresses).

When passing pointers into functions, the reference/address is given to the function. The pointers can be used to change the variables passed into the function.

**Global vs local**

Variables declared only exist at the level declared at or below (local).

Global variables are declared outside of functions (including main) and are accessible throughout the program. These have a separate section of memory allocated to them.

**Memory Assignment**

Machine code – code of the executed program.  
Globals – global variables  
Heap – free memory used up by program (and malloc). As the program requires more memory, heap increases in size.  
Stack – where functions & their variables are allocated memory. As more functions are used, stack requires more memory. Frees memory automatically as functions return  
Frame – chunk of memory allocated in stack every time you call a function. The “topmost” frame is the current stack, while those “below” it are on hold until the “topmost” frame is stopped and removed.  
If heap & stack collide (ie, too many functions or malloc) 🡪 creates a crash.

Heap overflow – overflow of the heap  
Stack overflow – overflow of stack  
Buffer overflow – overflow of buffer. Buffer is chunk of memory being used for some purpose. Occurs when there are logical errors in code.

**WEEK 5 – Data structures**

**Data Structures:**List – consecutive list of things  
Prepend – add to start of list  
Append – add to end of listQueue – list with first in first out property (FIFO)  
Enqueue – add to end of queue  
Dequeue – remove from first of queue  
Stacks – list with last in first out (LIFO)  
Push – add to top of stack  
Pop – remove from top of stack

**Nodes:**Data structures that have a value & a pointer.   
Pros – Dynamic allocation of memory (size can increase as needed).   
Cons - not contiguous (cannot use array notation), more memory usage, can only linear search/sort

Typedef struct node  
{

int number;  
 …  
 struct node \*next  
}  
node;

**Linked Lists**Lists that contain both values & pointers that link to the next element in the list. Can be dynamically allocated & pre/appended.Single-linked – Linked by single pointer in a long line.  
Double-linked – Linked by 2 pointers to create 2 dimensional structure.

**Trees:**When nodes are linked 2 dimensionally that typically grows downwards multiplicatively from a single “root” node.   
Binary search tree – form of a tree where each node has left & right pointers (2 pointers). Starting from a base “root” node, every node on the left is smaller & vice versa. This can be made using a simple recursive function.   
Pros – binary search can be used for O(logn).   
Cons – uses lots of memory, difficult to repair when one branch gets too long (lots of code).

**Dictionaries/Hash tables**Consists of Keys & Value pairs. Eg, name is key & value is phone number in a contact list  
Hashing – breaking down a list into multiple lists of smaller size 🡪 enables faster “real life” searching (not theoretically) at the cost of memory  
Hash function – the code/maths used in hashing  
Hash table – an array of linked lists designed to hash. Eg, A-Z array where different names are placed under appropriate array as linked lists.  
Setting hash table – node \*hash[n] 🡪 array of pointers to nodes (node (\*x)[n] 🡪 pointer to array of nodes)

**Tries**2 dimensional – each level contains an array of pointers which points to next character of search. NULL if the next character does not exist. Last character value is stored as true to show that it exists.  
Very fast but requires the most space & time to code.  
A computer screen shot of a black background

Description automatically generated  
typedef struct node  
{  
 char \*number  
 struct node \*children[n to hash]  
}  
node;

**Week 6: PYTHON**

**General:**

Object oriented language – objects can contain variables & functions  
Good for – string manipulation & data analysis  
Semicolons not needed  
Indentation important  
No ++. Use +=  
def to define functions – def fname(argument)  
No need for main function  
if want to start from main: put following in last line  
if \_\_name\_\_ == “\_\_main\_\_”:   
 main()

**Variables:**

Types automatically defined  
must set type when declaring

**Types:**

**str** – string  
str.capitalise()  
str.

**Lists:** array, but not set in size or type of variable  
nums = [x for x in range(n)]  
nums.insert(position, value)  
nums.append(value)

**Dictionary:** hash tables. Ie key & value   
dict = {key: value, …}  
dict[key] = value

**Tuple:** Ordered & immutable sets of data  
(‘value’, value, …)

**Object**  
class object():  
 def \_\_init\_\_(self, prop1, prop2, …): – initialisation function. Assign values to object properties  
 self.prop1 = prop1  
 self.prop2 = prop2  
def method(self, arg1, …):   
 #code  
var = object(prop1, prop2, …)  
var.method(arg1, …)

**Conditionals**

If condition:   
 #code  
elif condition:  
 #code  
else:  
 #code  
if/in – checks if specified value is in a string/list  
for element in list{start:end] – start &/ end optional. Start inclusive, end exclusive

Varname = value1 if condition else value2

**Comparisons**

and  
or  
not – reverse logical state  
True - capitalised  
False – capitalised  
is & is not – comparing memory addresses, ie – if referring to same object  
in & not in – checks for membership in sequence – str, list, tuple, dict  
!= - compares values of either side

**Loops**

While var < 100:   
 #code  
  
for var in var/number/range(): can iterate over str, list, tuple, dict   
 #code  
for var1, var2 in dict.items() – items() transforms dictionary into list. Order can change. Used to access values instead of keys  
 #print (var2)  
for/in – iterates over elements in list, iterates over characters in string  
for \_ condition: - when variable is not needed

**Functions:**

range([start], end, [increment])

print(‘string {varname}’, end  
print(‘string {[num]} string’.format(var, [var…])

len(variable) = length of str & lists

str/int(var) – change types

input(‘text’) – outputs texts and gets input

with open(“filename”) as varname:  
OR varname = open(“filename”, “r”/”w”) 🡪…-🡪 close(varname)

**Operations**

Num1 \*\* num2 – num1 ^ num2  
num1//num2 – num1 / num2 but rounded down (towards neg infinity)

**Import**

import lib – if importing whole library, functions must be called through lib.function()}  
from lib import function – imports specific function – can be called without ‘lib.’

**Week 7 – Working with data, SQL, CSV**

**CSV Library**csv.reader(filevar) – gets object that reads file. When iterated, returns list of a row contained with csvs. Can access using row[1]  
csv.DictReader(filevar) – reader but automatically assigns the initial/next row to be keys & returns dictionaries. When iterated, returns dictionary of a row contained by csvs under appropriate keys.

**Iterator –** Object that contains an iterable **Iterable –** Object that contains a countable number of values (discrete set) – list, tuple, dictionary  
itername = iter(iterable\_name, sentinel) – assigns an iterator of an iterable. Eg, For…in loop.   
**Sentinel –** Value that signifies end of sequence. Not needed  
next(itername, default) – returns the next iterable value. If default not given – raises StopIteration exception.  
**Getitem(index) –** invoked automatically using [] notation. Returns item from indexed value  
**In custom objects:  
Magic methods –** methods surrounded by “\_\_”. Can be used without callingDefine \_\_iter\_\_() AND (\_\_next\_\_() OR \_\_getitem\_\_() methods to use iter(), next(), getitem()

**Lambda:** 1 line function often used in iterators. No need to name – used if only needed once  
Lambda input: output

**SQL –** sequel/structured query language. Used to query relational databases – tables of rows & columns related to each other.  
**Data types:**INT, TINYINT, SMALLINT, MEDIUMINT, BIGINT  
DECIMAL, FLOAT  
BINARY, BIT, BLOB  
DATE, TIME, DATETIME, TIMESTAMP  
CHAR, VARCHAR – string of fixed/variable length. Must specify fixed/max length  
TEXT - strings  
ENUM – limited set of values  
**Additional constraints:** column datatype NOT NULL UNIQUE AUTO\_INCREMENT  
NOT NULL – column cannot be empty  
UNIQUE – every row must have unique value. Primary keys automatically have unique constraint  
AUTO\_INCREMENT – unique number automatically generated when new row entered in table  
**Index –** create indexes to retrieve data more quickly. Use on columns used to search frequently. Combinations of columns can be used to index  
CREATE INDEX index ON table (column1, …)  
CREATE UNIQUE INDEX index ON table (column1, …)  
**Generally adding/dropping constraints:**ALTER TABLE table ADD CONSTRAINT column CONSTRAINT (column1, …)  
ALTER TABLE table DROP INDEX column

**Functions  
CREATE –** CREATE TABLE table (column1 datatype, …, PRIMARY KEY (column). FOREIGN KEY (column) REFERENCES table(column)) **SELECT –** Get information out of table. SELECT column/s FROM table  
**SELECT DISTINCT –** selects only distinct values.SELECT DISTINCT column 1, … FROM table  
**JOIN –** get info from multiple tables. SELECT column/s FROM table1 JOIN table2 ON table1.column = table2.column

A diagram of a table

Description automatically generated with medium confidence  
**ALTER TABLE –** alter columns in existing table. ALTER TABLE table ADD/RENAME column datatype.   
**ALTER COLUMN –** alters datatype in existing column. ALTER TABLE table ALTER COLUMN column datatype **UPDATE –** Modify into in table. UPDATE table SET column = value WHERE conditional **INSERT –** adds information to table. INSERT INTO tablename (column1, …) VALUES (values1, …) **DELETE –** delete rows. DELETE FROM table WHERE conditional **DROP –** delete columns/tables. ALTER TABLE table DROP COLUMN column. DROP TABLE table

**Primary key:** Column/s with values that is unique to the row 🡪 for easy identification. Usually an integer 🡪can set to autoincrement. Can be created in CREATE or added/dropped via ALTER  
**Adding primary key -** ALTER TABLE table ADD PRIMARY KEY (column).  
**Drop primary key –** ALTER TABLE table DROP PRIMARY KEY

**Foreign key –** links to primary key in parent table. Used to protect links between tables. Tables with foreign key = child table. Can be created in CREATE or added/dropped via ALTER  
**Add foreign key -** ALTER TABLE table ADD FOREIGN KEY (column) REFERENCES table(column)  
**Drop foreign key –** ALTER TABLE table DROP FOREIGN KEY column

**Conditionals  
WHERE –** only the data meeting criteria are shown

**Operations:  
\* =** every column  
COUNT() – returns count  
LIKE – used in WHERE to search for specified pattern  
% - represents 0-many characters  
\_ - represents 1 character  
IN/NOT IN – multiple values in WHERE. SELECT column FROM table WHERE column IN/NOT IN (value1, …)  
UNION – combine result of 1+ SELECT statements. SELECT column/s FROM table1 UNION SELECT column/s FROM table2  
UNION ALL – allows duplicate values (which are automatically omitted in UNION)  
**ANY & ALL** – used to compare single column value vs range of values  
ANY – returns true if any queried values meet condition. SELECT column FROM table WHERE column operator ANY (SELECT column FROM table WHERE condition)  
ALL – returns true if all of query values meet condition. SELECT ALL column FROM table WHERE condition. SELECT column FROM table WHERE column operator ALL (SELECT…)  
**Changing order:   
ORDER BY –** ORDER BY column/s ASC/DESC. ASC default/not specified  
**GROUP BY –** groups rows that has same values into summary rows. **Eg – using count** 🡪 SELECT count(column1), column2 FROM table WHERE condition GROUP BY column/s

**WEEK 8 – HTTP, HTML, CSS, JS**

**Internet/network** – a number of servers & machines/PCs connected together, so information can be sent & received  
**Packets** – small bundles of information sent through network. If large (eg, videos) – separates fragments into different envelopes  
**Router** – decides on the path from point A to B, accounts for server load. Also works as a local DNS server to save most frequently accessed websites.  
**IP (Internet Protocol)** – protocol/conventions regarding address of machines on internet. #.#.#.# format uniquely identify each machine connected to internet. IPv4 – 32bits, IPv6 – 128bits.  
**TCP (Transmission Control Protocol)** – used to distinguish one type of service from another. Eg, 80=HTTP, 443=HTTPS is packaged in the packet, with an IP + port number. When packets are received, a reply is automatically sent to acknowledge reception  
**DNS (domain name system)** – collection of servers on internet to convert IP to domain names (eg, google.com)  
**Port -**   
**HTTP/HTTPS (HyperText Transfer Protocol/Secure)** – protocol scheme/service that runs on top of internet. Access server using TCP/IP to request information 🡪 server sends information to requesting machine 🡪 machine sends acknowledge.   
**Headers** – contains additional information in Key:Value pairs  
**Request headers** – contains information about resource to be fetched, or about client requesting resource.   
**Response headers** – contains information about the response. Eg, content-type:text/html.  
**.com/** - index route. Usually default when accessing website/server  
**.com/path** – access file stored at the address  
**localhost / 127.0.0.1** – used to reference own device   
**GET –** request type to get information from server. Information is sent through URL.   
**POST** – request type to post information to server. Information is not sent through URL  
**HTML (HyperText Markup Language)** – language used by browsers to show website  
**CSS** – language to change style of HTML. Often in own css file sent with HTML. Specific tags are selected via type, class, id, attribute to style. CSS is in Key:Value pairs  
**Javascript** – scripting language supported by browsers to make websites dynamic.

**WEEK 9 – Flask**

**Flask/Django** – python web framework to support backend of webpages.   
**Frontend** – visual aspects of website that users see & interact with  
**Backend** – structure, system, data & logic of website not seen by users.  
**Template** – basic HTML template that web frameworks can build different HTML files from, using code.  
**Cookies** – information stored in browser from websites to keep more long-term information about user that can be read by servers.   
**APIs** – how you can interact with a service to receive specific data.

**PYTHON/CS50P:**

**Python is written in C.**Everything in Python is an object of base Class PyObject – which contains a reference counter, typing and value. **Data type s** are just classes  
**Reference counter** – automatically incremented when assigned to variable or passed in function. De-incremented when finishing function, unassigned variable or put into a data structure.   
**Garbage collection** – python has 2 methods of garbage collection to manage memory:  
**Reference counting** – when reference count reaches 0, the memory for the PyObject is automatically deallocated. **Negatives** – inability to detect cyclic references.   
**Generational garbage collection** – a PyObject starts at 1st generation. When the number of objects exceeds a threshold for a generation, it executes garbage collection. Any objects that survive are put into the next generation. Each generation has different threshholds, which can be manually set  
  
Variables in python are names/pointers to a specific PyObject at a memory address.  
Assignment to a variable does not copy the PyObject into a new memory address. It points the variable to the same PyObject. IE, x = y 🡪 x is y == True  
**Mutable vs immutable** – most types of PyObjects are immutable and cannot be changed in value. When an operation to change value is done, a new PyObject with the resulting value is made at a new memory address and the variable is pointed to the new memory address.   
**Immutabl**e – int, float, str, tuple, etc.  
**Mutable** – [], {}. These are the only data types that are mutable, and therefore, when an operation (such as + or append) is done, it is added in place. The variable still points to the same PyObject at the same memory address.   
**Passing by object reference** – Immutable data types are passed by value, while mutable objects are passed by reference to functions.   
**Scope** – scope is defined by namespace. Each scope inherits namespace from the level above, but is unique to that scope. Therefore, if a local variable the same name as a global variable is defined in the scope, the local variable is used.

**Class** – a user defined data structure/type.   
**Instance/instance** – individual object/instance of certain class  
**Attributes** – data that is stored within an object/class  
**Properties** – special attributes with getters & setters to protect assignment (usually in cases where data validation is required for the attribute). When   
**Method** – a function contained within an object/class. Can reference the object that call it.   
instance.  
\_\_init\_\_ - special method that is called when an object/instance is first created.  
\_\_str\_\_, \_\_int\_\_ - called when another function expects this type of data input.  
**Getter –** special method to get the property of an object. @property. Automatically called when .property\_name value is accessed.   
**Setter –** special method to set the property of an object. @property\_name.setter. Automatically called when .property\_name is used with an = operator.   
Use \_ in front of property name in getter & setter functions to prevent variable name as .property\_name automatically calls the getter/setter. Technically the property that is stored is \_property\_name   
**Class variables –** variables outside of \_\_init\_\_ method. Shared by all instances of the class.   
**Class methods** – method that is associated to the class, not the instance/object. @classmethod. Useful when a class has children objects or method does not need a reference to an instance. Can reference its own class tho (convention is cls)  
**Static method –** a class method that does not get an implicit reference to self. Cannot access or modify class state. @staticmethod

**Decorators** – a function that takes a different function as an argument, and wraps code around the inputted function. @ used to decorate functions.

**Lambda functions** – anonymous functions used when a simple function is used once. Eg, sorting by a special order.