ANTONIUS' HANDBOOK

Useful Formulas, Constants, Units and Definitions
Volume II - Programmers Paradise
Version 0.043

Compiled by: Antonius William Torode
Written in: LaTeX

The original maintainer of this work is: Antonius Torode.

The current maintainer of this work is: Antonius Torode.

Published by Antonius Torode.

Hosted at: https://torodean.github.io/AHandbook.html

Github Repository: https://github.com/torodean/Antonius-Handbook-II

First Personal Release (Version 0.000):	June 2017
First Public Release (Version 1.000):	March 2024
Most Current Revision Date (Version 0.043):	March 15, 2024

Torode, A.
Antonius' Handbook.
Michigan State University –
Department of Physics & Astronomy.
2020, Graduate.
Volume II.
Version: 0.043

Preface

This document is a compilation of useful programming formulations, definitions, constants, and general information used throughout my own schooling and research as a reference while furthering education. It's purpose is to provide a complete 'encyclopedia' per se of various codes, syntax and significant ideas used often. The idea and motivation behind it is to be a quick reference providing easily accessible access to necessary information for either double checking or recalling proper formulations or algorithms for use in various situations due to my own shortcomings in matters of memorization. All the material in this document was either directly copied from one of the references listed at the end or derived from scratch. On occasion typos may exist due to human error but will be corrected when discovered.

The version number is updated every time the document is modified. This ensures that there is no two copies with different information and identical version numbers. The latest update date is automatically set to the current date each time the document is edited. Please refrain from distributing this handbook without permission from the original author/compiler. This book is formatted for printing.

For more information about this book or details about how to obtain your own copy please visit:

https://torodean.github.io/AHandbook.html

Disclaimer

This book contains codes, formulas, definitions, and theorems that by nature are very precise. Due to this, some of the material in this book was taken directly from other sources. This is only such in cases where a change in wording or codes could cause ambiguities or loss of information quality. Following this, all known sources used are listed in the references section.

This page intentionally left blank.
(Yes, this is a contradiction.)

Contents

Introduction

This handbook is a comprehensive collection of programming tips, snippets, notes, and more, curated by programmers for programmers. Whether you're a seasoned developer looking for quick references or a beginner seeking valuable insights, this handbook is designed to be your go-to resource. From C++ to HTML, LaTeX to Python, CMake, and beyond, Programmers Paradise covers a wide range of languages and topics. Its primary aim is to serve as a personal repository for quick retrieval of essential information, but its organized format and extensive content make it equally valuable for anyone in the programming community.

Linux

Linux is a broad subcategory that encompass a large family of free and open sourced operating systems. Installing, setting up, and using a Linux based operating system is the perfect way for anyone to gain knowledge, understanding, and practice of how a computer system truly works. Unlike the end user experience with Windows and Mac OS, Linux has a much higher capability for customization and a higher degree of freedom. With that said, Linux is not necessarily more user friendly to the new or average computer user, however it is free in most cases!

2.1 System Related Commands

Retrieve information and valid arguments for a command. This works with many commands.

```
COMMAND --help   # COMMAND must be a valid command such as cd, ls, etc...
```

Installing and updating packages using package managers such as **apt**, **yum**, or **dnf**.

```
sudo apt update  # Updates the list of available packages (APT)
sudo apt upgrade # Upgrades installed packages to their latest versions (APT)
sudo yum update  # Updates the list of available packages (Yum)
sudo yum upgrade # Upgrades installed packages to their latest versions (Yum)
sudo dnf update  # Updates the list of available packages (DNF)
sudo dnf upgrade # Upgrades installed packages to their latest versions (DNF)
```

Monitoring system resource usage using tools like **top**, **htop**, and **iotop**.

```
top   # Displays dynamic real-time information about running processes
htop  # An interactive process viewer
iotop # Displays I/O usage by processes in real-time
```

Changing directory via terminal via the **cd**command.

```
cd /directory # Changes the directory to the subdirectory /directory
cd ..         # Goes back one directory
```

Getting **current directory** via terminal

```
pwd
```

How to display the processes that are currently running.

```
ps aux
```

To search the results of a command for a string of characters one can use the grep command. For example:

```
ps aux | grep "firefox"
```

Restore power/battery icon if it disappears on a laptop.

```
/usr/lib/x86_64-linux-gnu/indicator-power/indicator-power-service &disown
```

Restore volume icon/control button if it disappears.

```
gsettings set com.canonical.indicator.sound visible true
```

Reset wifi services in case the connection gets lost.

```
sudo systemctl restart network-manager.service
```

Turn off LCD display.

```
xset dpms force off  // Turns off display.
```

Change or view the host name of a computer with the hostname file.

```
sudo nano /etc/hostname # Opens this file using nano for editing.
hostname                # Command to see what the current hostname is.
```

2.2 Remote Connections (SSH)

Accessing remote systems using **Secure Shell** (**SSH**).

```
ssh username@hostname     # Connects to the remote system using SSH
```

Transferring files securely between systems using **scp** or **sftp**.

```
# Copies a file from local to remote system
scp /path/to/local/file username@hostname:/path/to/remote/location

# Copies a file from remote to local system
scp username@hostname:/path/to/remote/file /path/to/local/location

# Initiates interactive SFTP session with remote system
sftp username@hostname
```

Accessing remote desktops using **X11 forwarding** and X terminal.

```
# Enables X11 forwarding for remote system
ssh -X username@hostname

# Opens a new X terminal window from remote system
xterm
```

Securely transferring files between systems using **rsync**.

```
# Synchronizes files and directories between local and remote systems
rsync -avz /source/path username@hostname:/destination/path
```

Accessing remote systems via graphical interface using tools like **VNC** or **XRDP**.

```
# Opens a VNC session to remote system
vncviewer hostname

# Opens a Remote Desktop Protocol (RDP) session to remote system
rdesktop hostname
```

Establishing persistent remote connections using tools like **tmux** or **screen**.

```
tmux new -s sessionname    # Creates a new tmux session
screen -S sessionname      # Creates a new screen session
```

2.3 Files and Storage

To find a file within a folder or its sub-folders, you can use the **find** command.

```
find -name "fileName.txt"  # Finds a file named fileName.txt
find -name "file*"  # Finds a file containing "file" in its name.
```

Copy a file or directory to a different computer

```
# To copy a file.
scp -v <File Path> username@computer:"<path to copy to>"

# To copy a directory.
scp -rv <File Path> username@computer:"<path to copy to>"
```

Show information about the file system on which each FILE resides, or all file systems by default.

```
df
```

Retrieve the Disk Usage (file sizes) of a directory or its contents.

```
du         # List the size of the subdirectories.
du -sh     # List the size of the directory in a human readable format.
du -ah     # Lists the size of all files in the directory.
```

List information about File(s) (in the current directory by default). You can specify the number of entries you want listed using the **head** and **tail** commands.

```
ls          # list all items in a directory
ls -1       # list all items in a directory (one item per line)
ls -lh      # list all items with size, owner, and date modified
ls -lhrt    # Useful ls output

ls -lhrt | head -4        # Outputs only the first four entires
ls -lhrt | tail -4        # Outputs only the last four entires
```

List all of the block devices (hence partitions) detected by the machine

```
lsblk
```

Mount and **unmount** a partition

```
sudo mount <DEVICE TO MOUNT> <MOUNT POINT>
sudo mount /dev/sdb1/ /mnt/       # example of mounting
sudo umount <DEVICE TO MOUNT> <MOUNT POINT>
sudo umount /dev/sdb1/ /mnt/      # example of mounting
```

To open a **pdf** via terminal, most generic desktop environments support

```
xdg-open filename.pdf
```

Creating backups using tools like **tar**, **rsync**, or **backuppc**.

```
# Creates a compressed tarball of a directory
tar -czvf backup.tar.gz /path/to/directory

# Synchronizes files and directories between two locations
rsync -avz /source/path /destination/path
```

2.4 Users and Groups

List all users

```
cut -d: -f1 /etc/passwd
```

Create a new user using the **useradd** command.

```
sudo useradd [options] <USERNAME>      # Creates a user.
sudo useradd -e 2016-02-05 <NAME>      # Creates a user that expites on a day.
sudo useradd <USERNAME> -G <GROUPNAME> # Adds a user to a group upon creation.
useradd --help                         # See full useradd options.
```

Change a users password using passwd.

```
passwd <USERNAME>
```

Change the user in terminal using the **su** command.

```
su - <USERNAME>
```

Add a user to the sudoers group

```
usermod -aG sudo <USERNAME>
```

Managing user permissions and file ownership using **chmod**, **chown**, and **chgrp**.

```
chmod 755 file         # Changes file permissions to read, write, and execute
   ↪ for owner, and read and execute for group and others.
chmod a+x file         # Changes file permissions to include execute
   ↪ permissions.
chown user:group file  # Changes the owner and group of the file.
chgrp group file       # Changes the group of the file.
```

2.5 Networking

The **ifconfig** command is for viewing IP configuration information and configuring network interface parameters.

```
ifconfig
```

The **traceroute** command is for printing the route that packets take to a network host.

```
traceroute
```

The **Domain Information Groper** is used to perform DNS lookups and display answers returned from the DNS servers.

```
dig
```

The **telnet** command connects the destination host:port via the telnet protocal. An established connection means connectivity between two hosts is properly working.

```
telnet
```

The **nslookup** command is for querying Internet domain name servers.

```
nslookup
```

The **netstat** command is used to review open network connections and open sockets.

```
netstat
```

The **nmap** command is used to check for opened ports on a server

```
nmap <SERVER NAME>
```

The **ifup** and **ifdown** commands are used to disable network interfaces.

```
# Enables an ethernet parameter
ifup <ETHERNET INTERFACE PARAMETER>
ifup eth0  # example: enables 'eth0'

# Disables an ethernet parameter
ifdown <ETHERNET INTERFACE PARAMETER>
ifdown eth0  # example: disables 'eth0'
```

Enable/Disable **IPv6**. This is only a temporary solution as it may turn itself back on after some time.

```
# Use these two commands to disable IPv6
sudo sysctl -w net.ipv6.conf.all.disable_ipv6=1
sudo sysctl -w net.ipv6.conf.default.disable_ipv6=1

# Use these two commands to re-enable IPv6
sudo sysctl -w net.ipv6.conf.all.disable_ipv6=0
sudo sysctl -w net.ipv6.conf.default.disable_ipv6=0
```

2.5.1 Redirecting program output

When outputting to a file, there is an append and an overwrite operator.

```
command > output.txt    // Writes output to output.txt
command >> output.txt   // Appends output to output.txt
```

To redirect all output from a program (including **stdout** and **stderr**), you can use

```
command -args > output.txt 2>&1
```

2.6 Shell/Bash Scripting

To create a shell script you must create a new text file and save it as a '.sh' file. The file should start with the directory to the proper shell which is generally the default below. The first line (starting with a **shebang** '#!') is not a comment, but instead is treated by Unix as "which shell do I use to run this code." In our case, the Bourne shell will be used [5]. Furthermore, to create a shell application that has parameters, with a help screen to explain those parameters, you can apply the following template.

```
#!/bin/sh
# This is a comment!

# This creates a method to print the usage of the script
usage()
{
    cat << EOF
purpose: This explains the purpose of the script
Usage:   $0 [opts]

OPTIONS:
    -h          Display this help message
    -f <file>   File to input
    -v          Boolean-type flag as parameter

EOF
```

```

# This parses the arguments input to the script. The ":" specifies a parameter is expected with the input flag.
while getopts "hf:v" OPTION; do
    case $OPTION in
        h)  usage           # Calls out usage method.
            exit 0          # Exits with error code 1 => success.
            ;;              # Ends the specified case
        f)  fileInput=$OPTARG
            ;;
        v)  verboseMode=1   # Sets verboseMode to true.
            ;;
        *)  echo "Invalid option entered: -$OPTARG" >&2
            usage
            exit 2          # Exits with error code 2 => fail.
            ;;
    esac                    # completes out case statement
done

# This checks if a flag was used.
if [[ $verboseMode ]]; then
    echo "Verbose mode is activated!"
elif [[ ! $verboseMode ]]; then
    echo "Verbose mode is disabled!"
fi
}
```

To print text one can use the **echo** command as follows.

```
#!/bin/sh
echo Hello World
echo "Hello World"
echo -e "In order to print newline characters, use the e option!\n"
```

To make a file executable, or change the permissions in general the **chmod** command can be used and is typically used as follows.

```
chmod a+x <SCRIPTNAME>.sh                    # Make a script executable.
chmod -R 775 filesToChangePermissionsOf.ext  # Change the permissions of some file.
```

To modify the ownership of files you can specify the owner and group of a file(s) using **chown**

```
chown username:group file.txt   # Change the owner and group of a file.
chown -R :group folder          # Change only the group of some folder and its sub-folders.
```

Shell script **variables** are created by use of the equal sign. spaces in lines containing variables need to be avoided. To reference a variable, the '$' character is used. Quotations are used to avoid ambiguities with spaces.

```
#!/bin/sh
MY_VARIABLE="Hello World"    # Creates a variable.
echo $MY_VARIABLE            # Prints the variable.
```

To use a variable within a terminal session, you can use **export** it to store it for that session.

```
export PATH="$PATH:/home/user/.local/bin/" # Appends the PATH variable with a
    ↪ string
```

The **touch** command can be used to create a new empty file.

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called ${USER_NAME}_file"

# The quotations prevent multiple files from being called to touch.
touch "${USER_NAME}_file"
```

To determine the total number of files of a given file patter, you can use the **wc** command to count the output of an ls command. To save the output of a script or command, you must enclose it within the correct elements of '$()' such as below.

```
totalFiles=$(ls -l $filePattern 2> /dev/null | wc -l)
```

To perform **arithmetic** within shell scripts, you must use double parenthesis.

```
firstVar=4
secondVar=7
addedVar=$(($firstVar+$secondVar))
```

To create a **for loop** in shell, you can do so like the following:

```
i=1
for file in $filePattern; do
    [[ ! -e $file ]] && continue
    echo "I see file number $i: $file"
    ((i++))
done
```

Bash provides a feature called "**brace expansion**," which allows you to generate arbitrary strings based on patterns using curly braces . While this feature is commonly used to generate sequences of numbers or letters, it can also be used in conjunction with command-line shortcuts to save time and improve efficiency.

```
# This will create directory1, directory2, ... and directory5.
mkdir directory{1..5}
```

2.7 Other

The **mkfifo** command is used to create a new named pipe. A pipe is used to store the output of one program to be used in another.

```
mkfifo namedPipe    # Creates a pip named "namedPipe".
ls > namedPipe      # Feeds the output of ls into namedPipe.
cat < namedPipe     # Feeds namedPipe into cat and displays the data from ls.
mkfifo namedPipe2 -m700  # Modifies the permissions of a created pipe.
```

2.7.1 emacs

Emacs is a powerful text editor. You can open a document through emacs using the following. The "-nw" flag indicates no GUI window should open (open in terminal).

```
emacs doc.txt -nw
```

To toggle **read-only mode** use C-x C-q.

Windows

3.1: Windows Key Combinations

Windows has various key combinations. These are used to do different things. For the purposes of this chart, the windows key will be represented by ⊞.

Key combination	Descriptions
⊞ + R	Opens a run window.
⊞ + M	Minimize all open windows.
⊞ + E	Opens the explorer window.
⊞ + UP	Minimize the currently opened window.
⊞ + F	Opens search for searching files and folders.
⊞ + P	Opens display projection settings.
Alt + Tab	Change between open windows.
CTRL + ALT + Delete	Provides user options such as changing password.
CTRL + SHIFT + ESC	Opens Windows Task Manager.

To access the Windows 7 "**God Mode**" which is essentially a collection of administrator and troubleshooting features, create a folder with the following name:

```
GodMode.{ED7BA470-8E54-465E-825C-99712043E01C}
```

To view system information, including RAM installed, graphics processor and more, run the following command

```
dxdiag
```

To view and manage the **services** that are running on a machine, you can access the services.msc application by opening a run window and entering

```
services.msc     # Opens the services running.
```

To view and modify **mouse** settings, such as sensitivity or speed, you can open a run window and enter

```
main.cpl         # Opens the mouse settings.
```

The **System File Checker** is a built-in tool that scans for and restores corrupted system files. You can run it from the Command Prompt with administrative privileges.

```
sfc /scannow
```

Mac

MacOS is Apple's proprietary Unix-based operating system tailored for Macintosh computers. It boasts a sleek graphical user interface (GUI) atop a Unix-like foundation, offering stability and security. Its software ecosystem encompasses a broad array of third-party applications accessible through the Mac App Store and other channels. Notable features include seamless integration with Apple's ecosystem, such as iCloud and iOS devices, robust security measures like Gatekeeper and FileVault for app and disk encryption, respectively, and advanced privacy controls. MacOS also provides essential productivity tools and development support, including Xcode and Terminal, making it an appealing platform for software engineers and developers.

4.1: Mac Startup Options

Mac has various startup features. To use them, hold the following keys down simultaneously upon startup as soon as you hear the startup chime:

Startup Keys	Descriptions
Command, R	Boot into OS X Recovery mode.
C	Boot to external device such as CD, DVD, or USB.
N	Netboot.
Shift	Safe Boot.
Command, V	Boot using verbose mode for comprehensive boot details.
Command, S	Single user mode.
Command, Option, P, R	Resetting the PRAM during boot.
T	Enable target disk mode.

C/C++

C++ is a powerful, high-level programming language renowned for its performance, flexibility, and extensive standard library. Developed by Bjarne Stroustrup in the early 1980s, C++ is an extension of the C programming language with added support for object-oriented programming (OOP) features, such as classes, inheritance, and polymorphism. C++ is widely used in systems programming, game development, embedded systems, and performance-critical applications due to its close-to-the-metal capabilities and efficient memory management. Its rich standard library provides developers with a wide range of functions and data structures for building robust and scalable applications. C++'s versatility allows developers to write code that is both low-level, allowing direct hardware access, and high-level, facilitating complex software design patterns. Despite its complexity, C++ remains a popular choice for developers seeking performance and control over their software projects.

5.1 Data Types

5.1: C Integer data types

This information is taken from **??**

C type	stdint.h type	Bits	Sign	Range
char	uint8_t	8	Unsigned	0 .. 255
signed char	int8_t	8	Signed	-128 .. 127
unsigned short	uint16_t	16	Unsigned	0 .. 65,535
short	int16_t	16	Signed	-32,768 .. 32,767
unsigned int	uint32_t	32	Unsigned	0 .. 4,294,967,295
int	int32_t	32	Signed	-2,147,483,648 .. 2,147,483,647
unsigned long long	uint64_t	64	Unsigned	0 .. 18,446,744,073,709,551,615
long long	int64_t	64	Signed	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
C type	IEE754 Name	Bits		Range
float	Single Precision	32		-3.4E38 .. 3.4E38
double	Double Precision	64		-1.7E308 .. 1.7E308

5.2 Basics of the Language

A basic C++ **hello world** program follows. This program includes the ¡iostream¿ header file, which provides input and output functionality. The main() function is the entry point of the program, and it simply outputs "Hello, World!" to the console using std::cout, which is the standard output stream. Finally, return 0; indicates successful program execution.

```
#include <iostream>

int main() {
    // Print "Hello, World!" to the console
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

The C++ main function is designed by default to pass in arguments when a program is ran. The argument passing is set up as follows.

```
int main(int argc, char *argv[])
{
```

```
    // argc would represent how many arguments were passed to the program.
    // argv[] is an array of each argument with the first element being the program name.
    for (int i=0;i<argc;i++) std::cout << argv[i] << std::endl; //Prints the arguments.
    return 0;
}

// So for example if I had HelloWorld.cpp as the above and ran using
// ./HelloWorld argument1 argument2 argument3
// we would get an output of
/some/file/path/HelloWorld
argument1
argument2
argument3
```

"The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as **address-of operator**. [1]

```
var = 314;        //Creates a variable and stores in in memory.
address = &var;   //Returns the memory address of the stored variable.
```

"**Pointers** are said to 'point to' the variable whose address they store." [1] Proceeding a pointer with the dereference operator (*), which can be read as 'value pointed to by' can be used to access a variable which stores the address of another variable (called a **pointer**).

```
pointer = *address; //Sets pointer to the value of the variable that address points to.
```

A pointer must be declared using the type of the data the pointer points to.

```
int * number;          //Creates number to point to an int.
char * character;      //Creates character to point to a char.
double * decimals;     //Creates decimals to point to a double.
int array [20];        //Creates an empty array with 20 elements.
number = &array[2];    //Sets number to point to the third memory slot of array.
cout << *number;       //Prints the value stored in array[2].
```

Incrementing pointers. "When adding one to a pointer, the pointer is made to point to the following element of the same type, and, therefore, the size in bytes of the type it points to is added to the pointer."

```
char *mychar;   //Creates mychar to point to a char.
short *myshort; //Creates myshort to point to a short.
long *mylong;   //Creates mylong to point to a long.

++mychar;       //Would increment to the next memory slot.
++myshort;      //Would increment two memory slots.
++mylong;       //Would increment four memory slots.

*p++   // same as *(p++): increment pointer, and dereference unincremented address
*++p   // same as *(++p): increment pointer, and dereference incremented address
++*p   // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

Within C++, you can use operations with **pointers to functions** which is typically used when calling a function with another function as a parameter. An example follows as [1]:

```
int addition (int a, int b){ return a+b; }
int subtraction (int a, int b){ return a-b; }

int operation (int x, int y, int (*functocall)(int,int)){
    int g;
    g = (*functocall)(x,y);
    return g;
```

```
}

int main (){
    int m,n;
    int (*minus)(int,int) = subtraction;  //minus is a pointer to a function that has two
        ↪ parameters of type int.

    m = operation (7, 5, addition);
    n = operation (20, m, minus);
    cout << n;
    return 0;
}
```

Templates can be used for defining classes that support multiple types.

```
template<class type>
class className{   //Creates a class named ClassName
    type a,b;    //Creates some variables of type
public:
    className(type val1, type val2) : a(val1), b(val2){};  //Constructor for className.
    type getMax(){ return a>b ? a:b; };
};
```

...Alternately, the above code can also be written as

```
template<class type>
class className{   //Creates a class named ClassName
    type a,b;    //Creates some variables of type
public:
    className(type val1, type val2){ a = val1; b = val2; };  //Constructor for className.
    type getMax();
};

template<class type>
type className<type>::getMax(){ return a>b ? a:b; }
```

To loop over boolean values, you can use

```
for (bool a : { false, true }) { /* ... */ }
```

5.2.1 Basic Input and Output

To output text via a terminal you can use:

```
uint32_t number = 0x123456         // A hexadecimal number.
std::string text = "Hello World!"; // A string.

std::cout << text << std::endl;    // std::endl is equivalent to the new-line character.
std::cout << std::hex << number;   // Prints a number in hexadecimal format.
```

To get input as a user in the type of a std::string, you can use:

```
std::string input = "";
std::cout << "Enter some text: ";
std::getline(std::cin, input);
```

5.2.2 Variable Types

Creating and using a vector.

```cpp
#include <vector>

int size1 = 5;
int size2 = 6;
uint32_t number = 0x345678;  //Creates a 32 bit unsigned integer and sets it in hexadecimal.

//Creates a vector named V1 containing int's with a size of 5 and sets each element to 0.
std::vector<int> V1(size1, 0);

//Creates a 2-D vector (vector containing vectors) of size 5x6 named V2 containing doubles ;
std::vector< std::vector<double>> V2(size1, std::vector<double>(size2, 0));

V1[0] = 8; //Sets the first element in V1 to 8.

V2[0][3] = 3.1415; //Sets the 4th element in the first row of V2 to 3.1415.
```

5.2.3 Class Structures

In C++ a **Class** is an object that can contain variables and functions all defined within the object to be used in various ways.

```cpp
// This creates a class named ParentClass.
class ParentClass{
    // The public members of a class are accessible to anything outside of the class.
    public:
        ParentClass();  // Constructor for the ParentClass.
        ~ParentClass(); // De-constructor for the ParentClass.
        int notSoSpecialInt = 13; // Creates an integer.
};

// This creates a class named ChildClass and inherits the public features of another class  ParentClass.
class ChildClass : public ParentClass {
    // The public members of a class are accessible to anything outside of the class.
    public:
        // Creates a public method to return notSoSpecialInt.
        int getNotSoSpecialInt() { return notSoSpecialInt; };
        // Creates a public method to return secretInt.
        int getSecretInt() { return secretInt; };

    // The protected members of a class are accessible to this class and any class that inherits this one.
    protected:
        void hello();      // Creates a protected method hello() that is not defined.

    // The private members of a class are only accessible to this class.
    private:
        int secretInt = 2; // Creates a private integer secretInt.
};

// This defines the hello method within the ChildClass class.
void ChildClass::hello() {
    std::cout << "Hello!" << std::endl;
```

```
}



int main() {
    ChildClass child;  // Creates an object of ChildClass named child.

    int number = child.secretInt();    // ERROR: This would not work because secretInt is
        ↪ private;
    int number = child.getSecretInt(); // SUCCESS: This works becase getSecretInt() is
        ↪ public.
    return 0;
}
```

5.2.4 Converting Between Types

5.2.4.1 std::string to *

To convert a **std::string** to an **integer** you can use the **stoi** function:

```
std::string text = "31415";
int number = std::stoi(text);
```

To convert a **std::string** to a **double** you can use the **stod** function:

```
std::string text = "3.1415";
double number = std::stod(text);
```

To convert a **std::string** to a **const char*** you can use the **c_str** function:

```
std::string str = "3.1415";
const char* chr = str.c_str();
```

To convert a **std::string** to a **long long int** you can use the **stoll** function:

```
std::string text = "123456789012345";
long long int number = std::stoll(text);
```

To convert a **std::string** to a **float** you can use the **stof** function:

```
std::string text = "3.1415";
float number = std::stof(text);
```

To convert a **std::string** to a **unsigned int** you can use the **stoul** function:

```
std::string text = "12345";
unsigned int number = std::stoul(text);
```

To convert a **std::string** to a **long double** you can use the **stold** function:

```
std::string text = "3.14159265358979323846";
long double number = std::stold(text);
```

5.2.4.2 * to std::string

To convert various types back to a **std::string**, you can use the **std::to_string** method:

```
int number_int = 31415;
std::string text_int = std::to_string(number_int);

double number_double = 3.1415;
```

```cpp
std::string text_double = std::to_string(number_double);

// For const char*, you can use the string constructor.
const char* chr = "3.1415";
std::string str_char(chr);

long long int number_ll = 123456789012345;
std::string text_ll = std::to_string(number_ll);

float number_float = 3.1415;
std::string text_float = std::to_string(number_float);

unsigned int number_uint = 12345;
std::string text_uint = std::to_string(number_uint);

long double number_ld = 3.14159265358979323846;
std::string text_ld = std::to_string(number_ld);
```

5.3 Mathematical Commands

5.3.0.1 Prime Numbers

A simple brute force method to determines if a number of type long is **prime** or not.

```cpp
bool isPrime(long num) {
    int c = 0;   //c is a counter for how many numbers can divide evenly into num
    if (num == 0 || num == 1 || num == 4) {
        return false;
    }
    for (long i = 1; i <= ((num + 1) / 2); i++) {
        if (c < 2) {
            if (num % i == 0) {
                c++;
            }
        } else {
            return false;
        }
    }
    return true;
}
```

5.3.0.2 Fibonacci Sequence

A simple brute force method for determining the **Fibonacci** numbers.

```cpp
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

5.3.0.3 Trigonometric Identities

To perform calculations using trigonometric identities, you first must include cmath and then do so as follows. These trigonometric functions from cmath can be used as floats, doubles, or long doubles.

```cpp
#include <cmath>                          // Needed at start of file.

float num = 0.05;                         // creating a number.
float numS = std::sin(num);               // Calculates the sin of the number
float numC = std::cos(num);               // Calculates the cos of the number
float numT = std::tan(num);               // Calculates the tan of the number
```

5.4 System Commands

Sleep

Make the thread **sleep** for some amount of time using the std::chrono to determine the duration [2].

```cpp
#include <thread>
#include <chrono>

// Makes the system sleep for 50 milliseconds.
std::this_thread::sleep_for(std::chrono::milliseconds(50));

// Makes the system sleep for 50 seconds.
std::this_thread::sleep_for(std::chrono::seconds(50));
```

On a **Windows** specific program this can be simplified by including the windows.h header

```cpp
#include <windows.h>

Sleep(50);   // Makes the system sleep for 50 milliseconds.
Sleep(5000); // Makes the system sleep for 50 seconds.
```

On a **Windows** specific program one can run a command directly from command prompt using the system function. The input variable to system is **const char***.

```cpp
#include <windows.h>

//system(const char* input)
system("DATE");   // Runs the DATE command from windows command prompt.
```

5.4.1 Simulate Key Strokes

5.4.1.1 Key Strokes on Windows

First the correct files must be included and an event must be setup.

```cpp
#define WINVER 0x0500
#include <windows.h>

INPUT ip;

ip.type = INPUT_KEYBOARD; // Set up a generic keyboard event.
ip.ki.wScan = 0; // hardware scan code for key.
ip.ki.time = 0;
ip.ki.dwExtraInfo = 0;
```

After this, functions can be setup to simulate various keys based on the specific key codes, two examples of such are

```cpp
void space() {
    // Press the "space" key.
```

```
    ip.ki.wVk = VK_SPACE; // virtual-key code for the "space" key.
    ip.ki.dwFlags = 0;    // 0 for key press.
    SendInput(1, &ip, sizeof(INPUT));

    // Release the "space" key.
    ip.ki.wVk = VK_SPACE;            // virtual-key code for the "space" key.
    ip.ki.dwFlags = KEYEVENTF_KEYUP; // KEYEVENTF_KEYUP for key release.
    SendInput(1, &ip, sizeof(INPUT));
    Sleep(50);
}

void one() {
    // Press the "1" key.
    ip.ki.wVk = 0x31;   // virtual-key code for the "1" key.
    ip.ki.dwFlags = 0;  // 0 for key press
    SendInput(1, &ip, sizeof(INPUT));

    // Release the "1" key.
    ip.ki.wVk = 0x31;                // virtual-key code for the "1" key.
    ip.ki.dwFlags = KEYEVENTF_KEYUP; // KEYEVENTF_KEYUP for key release.
    SendInput(1, \&ip, sizeof(INPUT));
    Sleep(50);
}
```

A similar method can be used to simulate mouse clicks. And example for left click follows

```
void leftclick() {
    INPUT ip={0};
    // left down
    ip.type = INPUT_MOUSE;
    ip.mi.dwFlags = MOUSEEVENTF_LEFTDOWN;
    SendInput(1,&Input,sizeof(INPUT));

    // left up
    ZeroMemory(&Input,sizeof(INPUT));
    ip.type = INPUT_MOUSE;
    ip.mi.dwFlags = MOUSEEVENTF_LEFTUP;
    SendInput(1,&Input,sizeof(INPUT));
    Sleep(50);
}
```

5.4.1.2 Key Strokes on Linux

First the correct files must be included and an event must be setup.

```
extern "C" {
    #include <xdo.h>
}

xdo = xdo_new(":1.0");
```

After this, functions can be setup to simulate various keys based on the specific key specifiers, two examples of such are

```
void space() {
    xdo_send_keysequence_window(xdo, CURRENTWINDOW, "space", 0);
}

void one() {
```

```
    xdo_send_keysequence_window(xdo, CURRENTWINDOW, "1", 0);
}
```

5.5 Compiler/Processor specific

The order of bytes within a binary representation of a number can be either **little endian** or **big endian**. In some cases, it is important to know this. Below is a function that will return the endianness of the machine you are compiling on.

```
bool is_big_endian(){
    union { uint32_t i; char c[4]; } bint = {0x01020304};
    return bint.c[0] == 1;
}

#if BYTE_ORDER == BIG_ENDIAN
    // Use big endian code here.
#endif

#if BYTE_ORDER == LITTLE_ENDIAN
    // Use little endian code here.
#endif
```

To define specific code to use on **Windows** vs **Linux** you can use the following

```
// This checks for windows or Cygwin.
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__) || defined
    ↪ _WIN32 || defined _WIN64 || defined __CYGWIN__
    // Windows only code here...
#elif __linux__
    // Linux only code here...
#endif
```

5.6 Advanced Coding Features

C++ allows you to perform **compile-time** evaluation of expressions using **constexpr** functions and variables. This feature enables you to compute values at compile-time rather than **runtime**, providing potential performance benefits and allowing for more flexible code optimization.

```
#include <iostream>

constexpr int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    constexpr int fib_10 = fibonacci(10);
    std::cout << "Fibonacci number at position 10: " << fib_10 << std::endl;

    return 0;
}
```

5.6.0.1 user-defined literals

C++ allows you to create **user-defined literals**, which are custom suffixes that can be applied to literal values to create objects of **user-defined** types. This feature enables you to create more readable and expressive code by introducing **domain-specific literals**.

```cpp
#include <iostream>

// Define a user-defined literal for converting meters to kilometers
constexpr long double operator"" _km(long double meters) {
    return meters / 1000.0;
}

int main() {
    // Use the user-defined literal to convert meters to kilometers
    long double distance = 5000.0_km;

    // Outputs "Distance in kilometers: 5"
    std::cout << "Distance in kilometers: " << distance << std::endl;

    return 0;
}
```

When using **user-defined literals**, it is up to the user to ensure that the units are consistent throughout the application. For example, the below defines celcius and farenheit temperatures. The celcius temperature has a conversion to itself, so all output temperatures will be in the units of celcius, whereas all other conversions will be converted to celcius.

```cpp
#include <iostream>

// Define a user-defined literal for Celsius
constexpr double operator"" _C(long double celsius) {
    return celsius;
}

// Define a user-defined literal for Fahrenheit
constexpr double operator"" _F(long double fahrenheit) {
    return (fahrenheit - 32.0) * 5.0 / 9.0;
}

int main() {
    // Use the user-defined literals to represent temperatures
    constexpr double freezing_point_C = 0.0_C;
    constexpr double boiling_point_C = 100.0_C;

    std::cout << "Freezing point of water in Celsius: " << freezing_point_C << std::endl;
    // Outputs "Freezing point of water in Celsius: 0"
    std::cout << "Boiling point of water in Celsius: " << boiling_point_C << std::endl;
    // Outputs "Boiling point of water in Celsius: 100"

    // Use the user-defined literals to convert Fahrenheit to Celsius
    constexpr double temperature_F = 212.0_F;
    constexpr double temperature_C = 98.6_F;

    std::cout << "Temperature in Fahrenheit: " << temperature_F << std::endl;
    // Outputs "Temperature in Fahrenheit: 100" (which is the celcius value)
    std::cout << "Equivalent temperature in Celsius: " << temperature_C << std::endl;
    // Outputs "Equivalent temperature in Celsius: 37"

    return 0;
```

```
}
```

5.6.0.2 templates

C++ provides a feature called "**template specialization**," which allows you to provide specialized implementations for specific **template** arguments. Template specialization is particularly useful when you need different behavior for certain template arguments while maintaining a generic implementation for others.

```
#include <iostream>

// Generic template
template <typename T>
struct MyTemplate {
    void print() {
        std::cout << "Generic template: " << typeid(T).name() << std::endl;
    }
};

// Specialized template for int
template <>
struct MyTemplate<int> {
    void print() {
        std::cout << "Specialized template for int" << std::endl;
    }
};

int main() {
    MyTemplate<double> obj1;
    obj1.print(); // Output: Generic template: d

    MyTemplate<int> obj2;
    obj2.print(); // Output: Specialized template for int

    return 0;
}
```

5.6.0.3 Function Pointers

C++ allows you to define and use "**function pointers**," which are **pointers** that point to functions rather than data. Function pointers enable you to treat functions as first-class citizens, allowing you to pass functions as arguments to other functions, store them in data structures, and call them dynamically at runtime.

```
#include <iostream>

// Function pointer type declaration
typedef int (*Operation)(int, int);

// Function to add two numbers
int add(int a, int b) {
    return a + b;
}

// Function to subtract two numbers
int subtract(int a, int b) {
    return a - b;
}

// Function to perform an operation using function pointer
```

```
int perform(Operation op, int a, int b) {
    return op(a, b);
}

int main() {
    // Define function pointers
    Operation addition = &add;
    Operation subtraction = &subtract;

    // Use function pointers to perform operations
    std::cout << "Addition result: " << perform(addition, 5, 3) << std::endl;
    std::cout << "Subtraction result: " << perform(subtraction, 5, 3) << std::endl;

    return 0;
}
```

Make/CMake

CMake is an open-source, cross-platform build system generator designed to facilitate the build process for software projects. Developed by Kitware in 2000, CMake simplifies the process of building, testing, and packaging software by providing a unified configuration language that abstracts away platform-specific details. CMake generates native build scripts for various platforms and compilers, including Unix Makefiles, Visual Studio solutions, and Xcode projects, allowing developers to maintain a single set of build instructions for multiple environments. With its modular and extensible architecture, CMake supports a wide range of project structures and dependencies, making it suitable for projects of all sizes and complexities. Its widespread adoption and active community support have established CMake as a standard build tool in the software development industry.

6.1 CMake

A poorly constructed and hard to follow yet fairly comprehensive example of how to use CMake and CMakeLists.txt files can be found at the following link (which is where much of the information in this section is derived from).

URL
https://cmake.org/cmake/help/latest/guide/tutorial/index.html

A basic project with CMake will contain an executable built from source code. To use CMake this project must contain a **CMakeLists.txt** file containing the following.

```
cmake_minimum_required( VERSION 3.14 )

# Set the project name and version.
project( ProjectName VERSION 1.001)

# Add an executable.
add_executable( Main main.cpp )

# Create a binary tree to search for include files.
target_include_directories( Tutorial PUBLIC "${PROJECT_BINARY_DIR}" )
```

To define and enable support for a specific **C++ standard**, you can use the following.

```
# Specify the C++ standard to use.
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

To add a **library** in a subdirectory and use that library in the main level, you must define where the library is in the main level CMakeLists.txt file.

```
# Add the LibraryName library.
add_subdirectory( LibraryDirectory )

# Links this Main file to the desired target library.
target_link_libraries( Main PUBLIC LibraryName )

# Add the binary tree to the search path for include files so that we will find
    ↪ LibraryName.h
target_include_directories( Main PUBLIC
    "${PROJECT_BINARY_DIR}"
    "${PROJECT_SOURCE_DIR}/LibraryDirectory"
)
```

In the directory containing the library, a CMakeLists.txt file must also exist and contain the following.

```
# Defines the file as a library.
add_library( LibraryName LibraryName.cpp)
```

To create **optional arguments** that can be turned on and off, one can use the **option** command.

```
# Creates a variable USE_MYLIBRARY and set it to on.
option( USE_MYLIBRARY "Use my library with this project" ON )
```

Variables like the above can be used as follows.

```
if( USE_MYLIBRARY )
    add_subdirectory(MathFunctions)
    list(APPEND EXTRA_LIBS LibraryName)
    list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/LibraryDirectory")
endif()

# add the executable
add_executable( Main main.cpp )

target_link_libraries( Main PUBLIC ${EXTRA_LIBS} )

# Add the binary tree to the search path for include files.
target_include_directories( Main PUBLIC
    "${PROJECT_BINARY_DIR}"
     ${EXTRA_INCLUDES}
)
```

The use of variables defined in CMake can be defined in the source code using

```
#cmakedefine USE_MYLIBRARY
```

You can also define a variable for use within a C++ file as follows

```
# CMakeList code here:
project(MIA VERSION 0.300)
add_definitions ( -DMIAVERSION=\"${VERSION}\" )

// C++ code then looks like this:
#ifdef MIAVERSION
    #define MIA_VERSION_VALUE MIAVERSION
#else
    #define MIA_VERSION_VALUE "Unknown"
#endif
std::string Configurator::ProgramVariables::MIA_VERSION = MIA_VERSION_VALUE;
```

To prevent needing a full relative file path in a cpp file include, you can use include the directory within the CMakeList file from that directory.

```
# CMake code:
include_directories(some/relative/path)

// C++ code
// #include "some/relative/path/file.hpp" // No longer needed
#include "file.hpp" // This replaces it
```

Git

Git is a distributed version control system designed for software development, enabling programmers to manage and track changes to their code base efficiently. Utilizing a decentralized architecture, Git allows developers to work collaboratively on projects, facilitating concurrent development and seamless merging of code changes. With features such as branching, tagging, and commit history, Git empowers programmers to maintain code integrity, experiment with new features, and revert changes when necessary. Its robust branching model facilitates the creation of feature branches for code development, enabling developers to isolate changes and merge them back into the main code base effortlessly. Overall, Git serves as an indispensable tool for programmers, offering a reliable and scalable solution for version control in software development projects.

7.1: Git Resources

Various git resources exist for use of or with git (an open source, distributed version-control system).

Description	Source
Main git website	https://git-scm.com/
git book	https://git-scm.com/book/en/v2
git reference	https://git-scm.com/docs/

7.0.1 Basic Git

To turn an existing directory into a git repository you can use the git **init** command,

```
git init   # Makes the current directory a git repository.
```

Git contains some basic **configuration** that can be set for all the local repositories.

```
# Sets the name you want attached to your commit transactions.
git config --global user.name "[name]"

# Sets the email you want attached to your commit transactions.
git config --global user.email "[email address]"

# Enables helpful colorization of command line output.
git config --global color.ui auto
```

Some basic commands used by git include cloning repositories with the **clone** (a local version of a repository, including all commits and branches) command, checking the **log** of previous changes, and checking the **status** of current files.

```
git clone https://github.com/torodean/Antonius-Handbook-II.git

git log         # Prints a list of the commits and their messages.
git log --stat  # Shows individual file changes along with the log.

git status      # Checks which state current files are in.
```

To **synchronize** your local repository with the remote repository.

```
git fetch    # Downloads all history from the remote tracking branches.
git merge    # Combines remote tracking branch into current local branch.
git pull     # A combination of git fetch and git merge.
```

To record changes to a git repository, you will primarily use the **add**, **commit** (a Git object, a snapshot of your entire repository compressed into a SHA), **diff**, and **push** commands. The **add** command can be thought of to "add precisely this content to the next commit".

```
git add -A              # Stages all files to be committed.
git add textFile.txt    # Stages a text file named 'textFile'.
git rm textFile.txt     # Removes a text file from staging.

git diff                # Shows unstaged changes.
git diff --cached       # Shows staged changes.

git commit              # Commits changes and asks for commit message.
git commit -m "text"    # Commits changes with a string as the commit message.
git commit -v           # includes the diff output into the commit.

git push                # Updates remote references using local references.
```

The **branch** (a lightweight movable pointer to a commit) command is used to create a new branch. The **checkout** command is used to change the working branch.

```
git branch                # Lists all branches.
git branch --merged       # See merged to current branches.
git branch -v             # See last commit on each branch.

git branch issue087       # Creates a branch issue087.
git checkout issue087     # Changes to the branch issue087.
git branch -d issue087    # Deletes the branch issue087.

git checkout -b issue087  # Does both of the above commands in one line.
```

The **merge** command is used to combine multiple branches after work on them is finished. To assist with merge conflicts, you can use **mergetool**.

```
git checkout master      # Changes to the master branch.
git merge issue087       # Attempts to merge master and issue087.

git mergetool            # Starts mergetool to assist with merge conflicts.
```

Using the fork command, one can create a copy of a repository owned by a different user.

```
git fork [URL]
```

Sometimes a file or type of file(s) are desired to be ignored by git and not push to the repository. This cane be done by creating a special file named **.gitignore**.

```
# Example .gitignore file. This file tells git to ignore the following types of files.
*.aux
*.log
*.synctex.gz
*.toc
*.o
```

7.0.2 Advanced Git

A useful tool is **stash** (code must be staged to be stashed), which lets you save code without making a commit [6][7].

```
git stash        # Makes a temporary local save of your repository.
git stash list   # Show's a list of stashes that have been made.
git stash apply  # Reapplies the content of a stash.
git stash branch # Carry over stashed commits to new branch.
```

```
git drop          # Used to remove stashes individually.
git stash clear   # Used to remove all stashes.

git checkout .    # Resets all uncommitted code.
```

The **reset** tool is used for accidental commits or reversing commits [6][7].

```
git reset   # Lets you modify your repository before doing a commit.
git reset --soft HEAD~NUM. # Resets the most recent $NUM of commits.
git reset --hard HEAD~NUM  # Erases your last $NUM of commits.
```

The **bisect** tool will present you with the details of a commit when compared with another. By referencing a good commit and a bad commit, git will traverse between the two and ask you which ones are good and which ones are bad and then you can display the differences after the process is finished [6][7].

```
git bisect         # Allows you to hunt for bad commits.
git bisect start   # Tells git that there is a bad commit.
git bisect bad     # Tells git which commit is bad.
git bisect good    # Tells git which commit is good.
git show           # Show the commit to indentify the issue.
```

The **rebase** tool is used to combine commits. The rebase tool can be dangerous as it could potentially permanently delete files. It is recommended to view the documentation before using it [6][7].

```
git rebase   # Allows for applying changes from one branch onto another.
```

To view and change the url that the git repository is using to update, you can use the git **remote** command.

```
git remote -v  # Lists the remote url of the repository.
git remote set-url origin <NEW-URL>  # Change the url of the repo.
```

If you've already pushed a commit to a remote repository, you can still modify the most recent **commit message** (assuming the branch you are pushing to is not protected). To do so, you use the **amend** flag with git **commit**. The **force-with-lease** flag checks if the remote branch has updates that do not exist locally.

```
# Update the commit message.
git commit --amend -m "New commit message"

# Force push to the branch.
git push --force-with-lease origin your-branch-name
```

7.0.3 Git Submodules

Git submodules are a feature in Git that allows developers to include one Git repository as a subdirectory within another Git repository. This enables the management of dependencies and external libraries within a project, allowing for modular and organized codebases. With Git submodules, developers can easily incorporate external code repositories into their projects while maintaining version control and tracking changes across multiple repositories. This provides a convenient way to reuse code, collaborate with external teams, and manage complex project dependencies effectively. However, it's important to note that working with Git submodules requires understanding of their workflow and potential complexities, making them a powerful but nuanced tool for managing project dependencies in software development.

For a project containing git **submodules**. When cloned (by default), the submodule repo's and containing files will not be cloned. Only a reference to the submodule commit will be cloned. To clone the submodule's files along with the repo, use

```
git clone --recurse-submodules <project-path>
```

If you have cloned a repository and would like to update submodule files after the fact, you can use

```
git submodule update --init --recursive
```

In a repository containing a submodule, the submodule is always pointing to a specific commit for that submodule. This commit may not be the latest of the submodule. To update all submodules to their latest commits, you can use the **remote flag**. After doing this to update submodules, changes will appear in the top level git reposity - as the submodule commit pointers have changed. These changes will need committed to the git repo.

```
git submodule update --init --recursive --remote
```

By default, each submodule will be in a detached **HEAD** state. To commit changes to a submodule, you must first change to a branch and then commit the changes. The top level repository contains a **.gitmodules** file that contains information on which branch the git repo will use for each submodule

```
# Example .gitmodules file
[submodule "submodule_name"]
    path = submodule_path
    url = submodule_url
    branch = submodule_branch
```

A Git command that allows you to run a given shell command in each submodule of your repository is the **foreach** command. This command is particularly useful when you're working with a repository that contains multiple submodules and you need to perform an operation across all of them.

```
git submodule foreach <command>
```

This can be used with any other git commands to perform the action on each submodule recursively.

```
# Pull changes for each submodule from the main branch.
git submodule foreach git pull origin main

# Get the status of each submodule.
git submodule foreach git status
```

7.1 Github.io

GitHub Pages is a static site hosting service provided by **GitHub** that allows users to host websites directly from their GitHub repositories. Users can create and publish websites for projects, documentation, personal portfolios, or blogs by simply committing and pushing HTML, CSS, and JavaScript files to a designated repository branch. GitHub Pages supports custom domains, HTTPS encryption, and various themes, making it a convenient and accessible platform for hosting static websites. Additionally, it integrates seamlessly with GitHub's version control system, enabling easy collaboration and versioning of website content.

URL
For a free tutorial on creating a github.io website, you can follow the following link:
`https://gitwebsitetutorial.github.io/`

LaTeX

LaTeX is a typesetting system widely used in technical and scientific documentation, renowned for its exceptional typesetting quality and support for complex mathematical equations and symbols. Programmers often leverage LaTeX to produce high-quality documents, reports, articles, and presentations with precise formatting and layout control. Its markup language allows for the creation of structured documents using plain text, enabling version control and collaboration using tools like Git. LaTeX's extensive package ecosystem offers additional functionality for generating bibliographies, tables, graphics, and custom document layouts, making it a preferred choice for programmers seeking precise and professional document production capabilities.

URL
For a repository of pre-made LaTeX templates, see:
`https://github.com/torodean/Antonius-Templates`

8.1 Hello World

To use this LaTeX document, you need to have a LaTeX distribution installed on your system, such as **TeX Live** or **MiKTeX**. Save the code in a file with a .tex extension, for example, hello_world.tex. Then, compile the document using a LaTeX compiler. This will generate a PDF output containing the "Hello, World!" message. You can view the PDF using any PDF viewer application. This example serves as a basic introduction to creating LaTeX documents and demonstrates the simplicity of LaTeX syntax for generating formatted text.

```
% Defines the type of document (article, report, book, etc.)
\documentclass{article}

\begin{document}    % Begins the document environment
Hello, World!       % Displays the text "Hello, World!"
\end{document}      % Ends the document environment
```

8.2 LaTeX Basics

To add **comments** in LaTeX, the percent sign is used on the line of the comment.

```
% This is a LaTeX comment
```

To include a **package** in LaTeX, the `\usepackage{}` command will import code from a LaTeX package. the package must be installed on the machine you are trying to import from.

```
\usepackage{multicol}
```

8.2.1 LaTeX Titles

In LaTeX, you can add **titles**, **subtitles**, and **authors** to your documents using specific commands. These titles are typically placed at the beginning of the document and help provide context and information about the content of the document. To add a title to your document, you can use the `\title{}` command. Inside the curly braces, you specify the title of your document. For example:

```
\title{My LaTeX Document}
```

If you want to include a **subtitle**, you can use the `\subtitle{}` command. For example:

```
\subtitle{A Beginner's Guide}
```

To specify the **author**(s) of the document, you can use the `\author{}` command. Multiple authors can be separated by the `\and` command. For example:

```
\author{John Doe \and Jane Smith}
```

Once you have defined the title, subtitle, and author(s), you can use the `\maketitle` command to generate the title page. This command should be placed after the `\begin{document}` command. The `\maketitle` command will format and display the title, subtitle, and author(s) according to the document class and style settings.

```
\maketitle
```

8.3 LaTeX elements

8.3.0.1 Quotations

For a simple, fancy, and nice looking quotation box, the following environment can be defined.

```
% Required packages.
\usepackage{fontawesome5}
\usepackage{tcolorbox}

% Create a custom color.
\definecolor{lightpurple}{RGB}{220,180,240}

% Create a custom quotation box environment.
\newenvironment{quotationbox}{
    \begin{tcolorbox}[
        colback=lightpurple!10,  % Very light purple background color of the box
        colframe=lightpurple!50, % Border color of the box
        arc=0pt,                 % Adjust the corner radius of the box
        boxrule=1pt,             % Border thickness
        title={\faQuoteLeft\ Quotation},  % The quotation icon and title
        fonttitle=\bfseries,     % Font style for the title
        coltitle=blue!50!black,  % Color for the title text
        colbacktitle=yellow!20,  % Background color for the title
        attach title to upper={\par},
        boxsep=0pt,              % Adjust the space between the content and box
        ]
    }{
    \end{tcolorbox}
}
```

8.3.0.2 URLs

For a simple, fancy, and nice looking url box, the following environment can be defined.

```
% Required packages.
\usepackage{fontawesome5}
\usepackage{tcolorbox}

% Create a custom url box environment.
\newenvironment{urlbox}{
    \begin{tcolorbox}[
        colback=blue!10,         % Very light green background color of the box
        colframe=blue!50,        % Border color of the box
        arc=0pt,                 % Adjust the corner radius of the box
        boxrule=1pt,             % Border thickness
        title={URL},             % The lightbulb icon and title
```

```
        fonttitle=\bfseries,    % Font style for the title
        coltitle=blue!50!black, % Color for the title text
        colbacktitle=yellow!20, % Background color for the title
        attach title to upper={\par},
        boxsep=0pt,             % Adjust the space between the content and box
        ]
    }{
    \end{tcolorbox}
}
```

For automatic **hyperlinking**, the hyperref package can be used.

```
\usepackage[unicode]{hyperref}

\url{www.duckduckgo.com}
```

C#

C# is a powerful, object-oriented programming language developed by Microsoft as part of the .NET initiative in the early 2000s. Designed for building robust, scalable applications, C# combines the productivity of modern, high-level languages with the performance and control of low-level languages. With its elegant syntax, rich standard library, and seamless integration with the .NET framework, C# enables developers to create a wide range of applications, from web and desktop applications to mobile and gaming platforms. Key features of C# include automatic memory management, type safety, and support for modern programming paradigms such as asynchronous programming and LINQ (Language-Integrated Query). Backed by a vibrant developer community and extensive documentation, C# continues to be a popular choice for building enterprise-grade software solutions across various industries.

The language C# is very similar to C++ and Java. All programming snippets listed in this section were tested and from a program created in Visual Studio 2013. Many of the functions written in this section depend on the Windows .Net application framework and may not function without that.

9.1 Useful Application Functions

Exit a program.

```
//These are needed at the beginning of the file.
using System;
using System.Windows.Forms;

//This exits the program
public static void exitLOLA(){
    if (System.Windows.Forms.Application.MessageLoop){
        System.Windows.Forms.Application.Exit();     // WinForms app
    } else {
        System.Environment.Exit(1);     // Console app
    }
}
```

The following will return the current Epoch time in seconds. This is useful for version control, random number generation, and more. Also, a demonstration of how ot set the current date as a string.

```
//Sets the current date as a string.
private static string today = System.DateTime.Today.ToString("d");

//Returns the current epoch time in seconds (time passed since January 1, 1970).
public static long getEpochTime(){
    var epoch = (DateTime.UtcNow - new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc)).
        ↪ TotalSeconds;
    return (long)epoch;
}
```

The following can be used to increase the size of a terminal window for a terminal application make in Visual Studio.

```
//Doubles the length of the output terminal window if the resolution on the computer
    ↪ permits it.
//Otherwise leaves it as the default.
public static void setScreenSize_double(){
    //determines the screen resolution to then determine the size of the output window.
    int origWidth = Console.WindowWidth;
    int origHeight = Console.WindowHeight;
    int height;
    int screenHeight = Screen.PrimaryScreen.Bounds.Height;

    if (screenHeight < 1080){
```

```
        height = origHeight;
    } else {
    height = origHeight * 2;
    }

    //int height = origHeight;
    Console.SetWindowSize(origWidth, height);
}
```

Get the directory path that the executable file is located in.

```
//returns the path that the program executable file is in.
public static string getProgramPath(){
    string path = System.IO.Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);
    return path;
}
```

9.2 Getting Windows System Information

Return the system name

```
//Returns the user defined system name.
public static string getSystemName() {
    systemName = Environment.MachineName;
    return systemName;
}
```

Determines and returns whether a processor is 32 or 64 bits and returns the number of bits as an int.

```
//Returns twhether the processor is 32 or 64-bit.
public static int getBits(){
    bool bitOS = Environment.Is64BitOperatingSystem;
    if (bitOS){
        bits = 64;
        return bits;
    } else {
        bits = 32;
        return bits;
    }
}
```

Returns the Full Operating System Name. Then, an alternate function to format the operating system name in a friendly manner.

```
public static string getOSFullName() {
    return new Microsoft.VisualBasic.Devices.ComputerInfo().OSFullName.ToString();
}

//Returns a 'friendly' string with the OS listed.
public static string getFriendlyOS() {
    OSversion = getOSFullName() + " " + getBits() + "-bit";
    return OSversion;
}
```

Returns the IPv4 address of a machine.

```
//Returns the IPv4 that the user is using.
public static string getIPv4address(){
    if (IPv4address != null){
```

```
        return IPv4address;
    }
    IPAddress[] ipv4Addresses = Array.FindAll(Dns.GetHostEntry(string.Empty).AddressList,
        ↪ a => a.AddressFamily == AddressFamily.InterNetwork);
    int i = 1;
    try{
        IPv4address = ipv4Addresses[i].ToString();
    }
    catch (System.IndexOutOfRangeException){
        i = 0;
        IPv4address = ipv4Addresses[i].ToString();
    }
    return IPv4address;
}
```

Determines CPU specs for a machine.

```
//Sets the CPU specs for the machine.
ManagementObject Mo = new ManagementObject("Win32_Processor.DeviceID='CPU0'");
uint speed = (uint)(Mo["CurrentClockSpeed"]);
string name = Mo["Name"].ToString();
Mo.Dispose();
int CPUspeed = Convert.ToInt32(speed);
string CPUmodel = name;
```

Determines the model of a PC.

```
string PCModel

System.Management.SelectQuery query = new System.Management.SelectQuery(@"Select * from
    ↪ Win32_ComputerSystem");
using (System.Management.ManagementObjectSearcher searcher = new System.Management.
    ↪ ManagementObjectSearcher(query))

foreach (System.Management.ManagementObject Mo in searcher.Get()){
    Mo.Get();
    string Model = Mo["Model"].ToString();
    //System.Console.WriteLine("{0}{1}", "...System Model: ", Mo["Model"]);
    PCModel = Model;
}
if (PCModel == "System Product Name"){
    PCModel = "unknown";
}
```

Returns the installed RAM in a machine. Multiple methods are listed which give varied results depending on the environment.

```
//This returns an estimate of the ram installed on a machine.
//It is keyed to take the total bytes of RAM and cnovert them to the nearest 2^n value.
//This is the old function to determine RAM capacity. getInstalledRAM() returns a more
    ↪ accurate value.
public static ulong getTotalPhysicalMemoryInBytes(){
    return new Microsoft.VisualBasic.Devices.ComputerInfo().TotalPhysicalMemory;
}
public static ulong getTotalVirtualMemoryInBytes(){
    return new Microsoft.VisualBasic.Devices.ComputerInfo().TotalVirtualMemory;
}

//Returns the physically installed RAM.
public static ulong getInstalledRAM(){
```

```
    string Query = "SELECT Capacity FROM Win32_PhysicalMemory";
    ManagementObjectSearcher searcher = new ManagementObjectSearcher(Query);

    UInt64 Capacity = 0;
    foreach (ManagementObject WniPART in searcher.Get()){
        Capacity += Convert.ToUInt64(WniPART.Properties["Capacity"].Value);
    }

    return Capacity;
}
```

Java

Java is a versatile, object-oriented programming language renowned for its portability, scalability, and security features. Developed by Sun Microsystems in the mid-1990s, Java has since become a cornerstone of enterprise and web application development. One of Java's key strengths is its "write once, run anywhere" philosophy, facilitated by the Java Virtual Machine (JVM), which allows Java programs to run on any platform that supports the JVM, regardless of the underlying hardware or operating system. Java offers a rich set of libraries and frameworks for building a wide range of applications, from desktop and mobile apps to enterprise systems and web services. Its strong emphasis on object-oriented principles, robust memory management, and extensive community support make Java a popular choice for both beginner and experienced developers alike.

10.1 Java Basics

A basic Java **hello world** program follows. In this program, we define a class named HelloWorld. Inside this class, we have a main method, which is the entry point of the program. Within the main method, we use System.out.println() to print "Hello, World!" to the console.

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         // Print "Hello, World!" to the console
4         System.out.println("Hello, World!");
5     }
6 }
```

HTML

HTML (Hypertext Markup Language) is a fundamental language for creating and structuring web pages. It consists of tags that define the structure and content of a document, allowing developers to organize text, images, links, and other media elements. HTML documents are hierarchical, with a tree-like structure where each element is nested within others, forming a Document Object Model (DOM). Developers can manipulate the DOM using JavaScript to dynamically update content and interact with users. HTML5, the latest version of HTML, introduced new features like semantic elements (e.g., <header>, <footer>) and multimedia support (e.g., <video>, <audio>), enhancing the capabilities for building modern web applications. Understanding HTML is essential for web developers to create accessible, well-structured, and responsive websites.

11.1 HTML Basics

The below HTML is a basic example of an HTML document structure (**hello world**). It should be saved with a '.html' extension. It starts with the document type declaration (<!DOCTYPE html>), indicating that it is an HTML5 document. Inside the <html> element, the **lang** attribute specifies the language of the document (English in this case). The document contains a <head> section where **metadata** and resources for the document are defined. In this example, it includes the **character set** (<meta charset="UTF-8">), and the title of the document (<title>Hello, World!</title>). The main content of the document is contained within the <body> element. In this example, it includes a heading (<h1>) displaying the text "Hello, World!", followed by a paragraph (<p>) with the text "This is a simple HTML file."

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello, World!</title>
</head>
<body>
    <h1>Hello, World!</h1>
    <p>This is a simple HTML file.</p>
</body>
</html>
```

HTML **comments** are enclosed within '<!–' and '–>' tags. Anything between these tags is treated as a comment and is ignored by the web browser when rendering the page. Comments can span multiple lines and can be placed anywhere within the HTML document. Here's an example of an HTML comment:

```
<!-- This is a comment. It will not be displayed in the browser. -->
```

To add a **stylesheet** to an HTML document (for css or other elements), add a link within the head element.

```
<head>
    <!-- Add icon library -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome
        ↪ /4.7.0/css/font-awesome.min.css">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="css/vendors/bootstrap/bootstrap.min.css">

    <!-- main css -->
    <link rel="stylesheet" href="css/styles.css">
</head>
```

The **div** element is a generic container that allows you to group and style content together. It's commonly used for layout purposes and to apply styling or scripting to multiple elements at once.

```
<div id="container">
    <h2>This is a div container</h2>
    <p>This is some content inside the container.</p>
</div>
```

The **span** element is an inline container used to mark up a part of text or a part of a document. It's often used to apply styles to specific parts of text.

```
<p>This is a <span style="color: red;">highlighted</span> word.</p>
```

The **img** element is used to embed images in an HTML document.

```
<img src="image.jpg" alt="Description of image">
```

The **a** element is used to create hyperlinks to other web pages, files, locations within the same page, or email addresses.

```
<a href="https://www.example.com">Link to Example Website</a>
```

11.2 Useful HTML scripts

CSS

CSS (Cascading Style Sheets) is a styling language used to control the visual presentation of HTML and XML documents. It allows developers to define styles for elements such as colors, fonts, layout, and spacing, separating the content from its presentation. CSS works by selecting elements in a document and applying styles to them based on rules defined by the developer. It offers various selectors and properties to target specific elements or groups of elements, enabling fine-grained control over the appearance of a web page. CSS can be applied inline within HTML elements, embedded in the ¡style¿ tag in the document head, or linked externally to the HTML file. With CSS, developers can create visually appealing and responsive web designs, improving user experience and accessibility across different devices and screen sizes. Understanding CSS is essential for front-end developers to create attractive and well-designed web interfaces.

URL
For a free visual reference to css, you can use the following link:
`https://cssreference.io/`

12.1 CSS Basics

CSS supports **comments** using the `/* */` syntax. Comments are ignored by the browser.

```
/* This is a CSS comment */
```

CSS style rules consist of a **selector** and a **declaration** block. The selector targets one or more HTML elements, and the declaration block contains one or more property-value pairs that define the styles to apply. An example of a CSS rule follows. In this example, the selector `h1` targets all `<h1>` elements, and the declaration block sets the text color to blue, font size to 24 pixels, and font weight to bold.

```
h1 {
    color: blue;
    font-size: 24px;
    font-weight: bold;
}
```

CSS uses **selectors** to target specific HTML elements for styling. There are various types of selectors, including element selectors, class selectors, ID selectors, and more. For example, to style all paragraphs with a class of "intro," you can use the following. This rule will italicize the text of all paragraphs with the class "intro."

```
p.intro {
    font-style: italic;
}
```

The **color** property sets the text color. It can be specified using color names, RGB values, HEX codes, or HSL values.

```
/* Using RGB value */
body {
    color: rgb(51, 51, 51);
}

/* Using HEX value */
body {
    color: #333;
}

/* Using color name */
body {
    color: darkgray;
}
```

To modify the **background** look, you can use the following properties:

```
body {
    background-color: #f0f0f0; /* Light gray background color */
    background-image: url("background.jpg"); /* Background image */
    background-repeat: no-repeat; /* Do not repeat the background image */
    background-size: cover; /* Cover the entire background */
    background-position: center; /* Center the background image */
    background-attachment: fixed; /* Fix the background image */
}
```

PHP

PHP (Hypertext Preprocessor) is a server-side scripting language commonly used for web development. It enables developers to create dynamic and interactive web applications by embedding PHP code within HTML files. PHP code is executed on the server, generating HTML content that is then sent to the client's web browser. PHP offers a wide range of features for handling form data, interacting with databases, managing sessions and cookies, and performing file operations. It also supports object-oriented programming, allowing developers to create reusable and modular code. PHP integrates seamlessly with various web servers and database management systems, making it a versatile choice for building dynamic websites and web applications. Understanding PHP is essential for back-end developers to implement server-side functionality and deliver dynamic content to users.

13.1 PHP Basics

A basic PHP **hello world** program follows. In this PHP script, we use echo to output "Hello, World!" to the browser. The PHP tags ¡?php ... ?¿ indicate the beginning and end of PHP code blocks.

```
<?php
// Print "Hello, World!" to the browser
echo "Hello, World!";
?>
```

JavaScript

JavaScript, often abbreviated as JS, is a dynamic, lightweight programming language primarily used for client-side web development. Developed by Brendan Eich at Netscape in the mid-1990s, JavaScript has evolved into a versatile and essential tool for creating interactive and dynamic web content. Unlike Java, which is primarily used for server-side programming, JavaScript is executed on the client's browser, enabling developers to enhance user experiences with features such as dynamic content updates, form validation, and interactive elements. JavaScript is supported by all modern web browsers and is often used in conjunction with HTML and CSS to create modern, responsive web applications. Its flexibility, simplicity, and widespread adoption have made JavaScript an integral part of web development, powering everything from simple webpage animations to complex single-page applications.

Python

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. With its elegant syntax and dynamic typing, Python facilitates rapid development and prototyping across various domains, including web development, data analysis, artificial intelligence, and scientific computing. Its extensive standard library and vibrant ecosystem of third-party packages provide comprehensive support for a wide range of tasks, from basic scripting to complex software development. Python's object-oriented and functional programming paradigms, coupled with its strong community support and cross-platform compatibility, make it an ideal choice for both beginners and experienced developers seeking efficient and expressive solutions to their programming challenges. Its emphasis on code readability and simplicity distinguishes Python from other languages, promoting maintainability and collaboration in software projects.

The official python documentation can be found at the following links

```
# Documentation for version 3+
https://docs.python.org/3/
# Documentation for version 2+
https://docs.python.org/2/
```

15.1 Basics of Python

15.1.1 Imports and libraries

Import floating point division which allows python 2 compatibility when using division with doubles. Include this at the beginning of the script.

```
from __future__ import division
```

On Linux, there's a site-packages folder that imported libraries are all in. These are located in the following directories for python versions 2.7 and 3.6 respectively.

- /usr/lib/python2.7/site-packages
- /usr/lib/python3.6/site-packages

I believe you only have specify the path from that folder on. For example, an app that uses the import program.script.interface as interface will have the interface.py file installed to usr/lib/python3.6/site-packages/program/scripts.

15.1.2 Program parameters

You can add input parameters for a python script/program using the argparse package.

```
import argparse

parser.add_argument(
        '-v',                           # Creats a parameter with -v
        '--verbose',                    # Creates args.verbose parameter
        required=False,                 # Sets the parameter to not required.
        default=False,                  # Sets default value
        action='store_true',            # Sets value to true if supplied
        help="Enables verbose mode.")   # Sets help message

parser.add_argument(
        '-i',                           # Creates a parameter with -i
        '--input',                      # Creates args.input parameter
        required=True,                  # Sets the parameter to be required
        default="potato",               # Sets default value to "potato"
```

```
        help="An input string to use")  # Sets a help message

if args.input is not none:
    print("Input value is {0}".format(args.input)) # prints the input value.

if args.verbose:
    print("Verbose is on!")
```

15.1.3 Methods, Functions, and Classes

To define a **function** in Python, you use the **def** keyword followed by the function name and parameters, if any. You can also include a docstring to provide documentation for the function. In this example, the `greet()` function takes a `name` parameter and returns a greeting message.

```
def greet(name):
    """
    This function greets the user.
    :param name: The name of the person to greet
    :return: A greeting message
    """
    return f"Hello, {name}!"
```

Methods are functions that are associated with objects. They are defined within the scope of a class and are accessed through instances of the class. In this example, the `bark()` method of the `Dog` class prints a bark message when called.

```
class Dog:
    def bark(self):
        """
        This method makes the dog bark.
        """
        print("Woof!")
```

Classes in Python allow you to define your own data types with custom attributes and methods. In this example, the `Rectangle` class represents a geometric rectangle. It has an **__init__**() method to initialize its width and height attributes and an `area()` method to calculate its area.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

To use the `Rectangle` class, you can create an instance of it by passing the width and height values as parameters to the constructor:

```
# Constructing a Rectangle object
rectangle1 = Rectangle(5, 10)

# Calculating the area of the rectangle
area = rectangle1.area()

print("The area of the rectangle is:", area)
```

15.1.4 String manipulation

To split a string by a **delimiter** you can use the **split**() method. To remove any leadig or trailing **whitespace** from a string, you can use the **strip**() method. Additionally, you can use **lstrip**() to remove leading whitespace and **rstrip**() to remove trailing whitespace.

```
stringValue = "Test; string "

firstPart = stringValue.split(';')[0]    # Stores "Test"
secondPart = stringValue.split(';')[1]   # Stores " string "
noWhiteSpace = secondPart.strip()        # Stores "string"
leftWhiteSpace = secondPart.lstrip()     # Stores "string "
rightWhiteSpace = secondPart.rstrip()    # Stores " string"
```

The opposite of splitting strings is joining them. You can use the **join**() method to concatenate a sequence of strings into a single string, using a specified delimiter.

```
words = ['Hello', 'world', '!']

# Join the words with a space delimiter
sentence = ' '.join(words)    # Stores "Hello world !"
```

Python provides methods to convert the case of strings. You can use **lower**() to convert a string to lowercase and **upper**() to convert it to uppercase.

```
text = "Hello, World!"

# Convert the string to lowercase
lowerCaseText = text.lower()    # Stores "hello, world!"

# Convert the string to uppercase
upperCaseText = text.upper()    # Stores "HELLO, WORLD!"
```

To insert variables into a string, you can use the **format**() method or by using an **f-string** (introduced in python3.6).

```
a = 5
b = 6
sum = a + b
equation = "{0} + {1} = {2}".format(a, b, sum)
f_equation = f"{a} + {b} = {sum}"

print(equation)   # prints "5 + 6 = 11"
print(f_equation) # prints "5 + 6 = 11"
```

15.1.5 Arrays and lists

To check if any value in one array or list is present in another array or list, you can use the **any**() function along with a generator expression. In this example, the **any**() function iterates over each item in `list2` and checks if it is present in `list1`. If any common elements are found, the condition evaluates to `True`, and the corresponding message is printed.

```
list1 = ["apple", "banana", "orange"]
list2 = ["orange", "grape", "pear"]

if any(item in list1 for item in list2):
    print("Common elements found between list1 and list2")
else:
    print("No common elements found")
```

You can concatenate two or more lists using the **+ operator** or the **extend**() method. Both methods produce the same result: a new list containing all the elements from the original lists in the specified order.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Using the + operator
concatenated_list = list1 + list2   # Stores [1, 2, 3, 4, 5, 6]

# Using the extend() method
list1.extend(list2)   # Modifies list1 to [1, 2, 3, 4, 5, 6]
```

To find the number of elements in a list, you can use the built-in **len**() function.

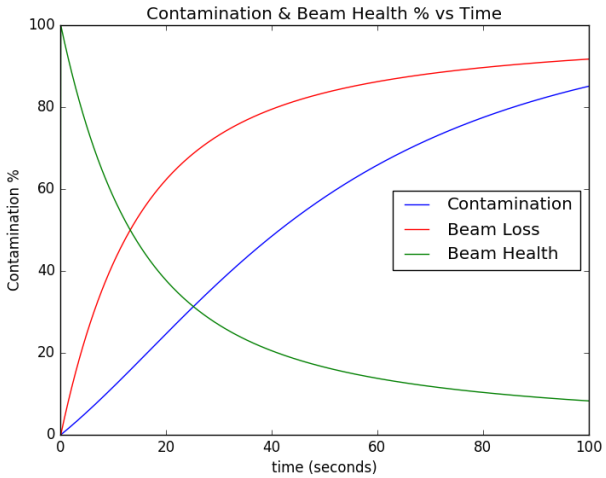
```
my_list = [10, 20, 30, 40, 50]

# Find the length of the list
list_length = len(my_list)   # Stores 5
```

15.1.6 Plotting and Graphs

A nicely formatted plot with a legend using the pylab package.

```
import pylab as plt #Imports the correct packages for plotting.

plt.title('Contamination & Beam Health % vs Time')      # Creates a title.

plt.plot(t, Contamination, '-b', label='Contamination') # Plots Contamination in blue.
plt.plot(t, Beam_loss, '-r', label='Beam Loss')     # Plots Beam_loss in red.
plt.plot(t, Beam_health, '-g', label='Beam Health') # Plots Beam_health in green.

plt.xlabel("time (seconds)")   # Creates a x-axis label
plt.ylabel("Contamination %")  # Creates a y-axis label

plt.legend(loc='center right') # Creates a legend with the labels set above.
# Other locations include upper/lower/center left/right

plt.show() # Displays plot.
```

This code would display a graph such as the one below such that the proper values are input.



A nicely formatted plot with a legend using the matplotlib package.

```
import matplotlib as plt    # Imports the correct packages for plotting.

fig = plt.figure(dpi=1200)  # Increase resolution of the plot.
plt.scatter(x, y, s=0.1)    # Plot x data vs y data with a dot size of 0.1.
fig.suptitle(title, fontsize=12) # adds a title to the figure.

plt.xlabel("x axis label")  # Creates a x-axis label.
plt.ylabel("y axis label")  # Creates a y-axis label.

manage = plt.get_current_fig_manager()
manage.full_screen_toggle() # Makes the plt full screen.

plt.show()                  # Displays plot.
plt.savefig("fileName.png") # Saves the figure as an image
```

15.2 Pytest

Pytest is a powerful and popular testing framework for Python that simplifies the process of writing and executing tests. It offers a straightforward syntax and extensive features for writing test cases, including assertions, fixtures, parameterization, and test discovery. Pytest provides robust support for testing different types of applications, including web applications, APIs, and command-line utilities. It promotes efficient testing practices by emphasizing readability, scalability, and flexibility, making it a preferred choice for many Python developers.

To run a python test file using pytest, you can call pytest directly via a terminal.

```
pytest -vv test_my_module.py
```

To write tests for a function, you must import that function and write a test for it. The **assert** is a basic test keyword is used to check if a statement is true of false.

```
# Code implementation in a Python file (e.g., my_module.py)

def add(x, y):
    """Function to add two numbers."""
    return x + y
```

```
# Test case using Pytest (e.g., test_my_module.py)

import pytest
import my_module

def test_add():
    """Test case for the add function."""

    # Test with positive integers
    assert my_module.add(2, 3) == 5

    # Test with negative integers
    assert my_module.add(-2, -3) == -5

    # Test with zero
    assert my_module.add(0, 0) == 0

    # Test with one positive and one negative integer
    assert my_module.add(5, -3) == 2
```

A pytest **fixture** can be used to create a **temporary directory** for testing.

```python
import pytest

@pytest.fixture
def temp_directory():
    """
    Fixture to create a temporary directory for testing.

    This fixture creates a temporary directory using tempfile.mkdtemp().
    The temporary directory path is provided to the test functions, and after testing,
    the temporary directory is removed to clean up resources.

    Yields:
        str: Path of the temporary directory.
    """
    # Create a temporary directory for testing
    temp_dir = tempfile.mkdtemp()
    # Provide the temporary directory to the test for its duration.
    yield temp_dir

# Call a method using the temp_directory() method.
def test_method(temp_directory):
    print(temp_directory) # Prints the temp directory path
```

A pytest **fixture** can be used to return an instance of a class.

```python
import pytest
from mmorpdnd import MMORPDND

@pytest.fixture
def mmorpdnd_instance():
    """
    Fixture to provide an instance of the MMORPDND class for testing.
    """
    return MMORPDND()
```

You can define a list of test **parameters** to be input into a test via the **@pytest.mark.parametrize** decorator, allowing multiple input-output pairs to be tested with the same test function.

```python
@pytest.mark.parametrize("input_string, expected_output", [
    ("'6 (barbarian 3, rogue 3)'", 6),
    ("'12 dwarfs and 3 elves'", 12),
    ("'No numbers here!'", None),
    ("'Only one number: 42'", 42),
    ("'Negative number: -10'", -10),
    ("'Integers: 1 2 3'", 1),
    ("'999'", 999),
])
def test_extract_first_integer(input_string, expected_output):
    """
    Test case to verify the behavior of extract_first_integer function.
    """
    assert extract_first_integer(input_string) == expected_output
```

15.3 Python tk GUI

Python **Tkinter** (**Tk**) is a built-in **GUI** (Graphical User Interface) toolkit that allows developers to create desktop applications with a graphical interface in Python. Tkinter provides a set of widgets (such as buttons, labels, and entry

boxes) and tools to arrange them within windows and frames. It is based on the Tk GUI toolkit originally developed for the Tcl programming language. With Tkinter, developers can create cross-platform applications that run on Windows, macOS, and Linux, making it a popular choice for simple to moderately complex desktop applications in Python. To install Tk on linux (APT), the following command can be used.

```
sudo apt-get install python3-tk
```

A basic **hello world** gui written in Tk follows:

```
import tkinter as tk

def say_hello():
    label.config(text="Hello, World!")

# Create the main application window
gui = tk.Tk()
gui.title("Hello, Tkinter!")

# Create a label widget
label = tk.Label(gui, text="Click the button to say hello!")
label.pack(pady=10)

# Create a button widget
button = tk.Button(gui, text="Say Hello", command=say_hello)
button.pack()

# Run the Tkinter event loop
gui.mainloop()
```

To change the **geometry** of a Tk gui window, the following can be used:

```
gui = tk.Tk()
gui.geometry("300x470")
```

To change the **icon** of a Tk gui window, the following can be used:

```
# Load icon image
icon = PhotoImage(file=f'{root_dir}/img.png')
# Set icon image
gui.tk.call('wm', 'iconphoto', self.gui._w, icon)
```

15.4 Useful Methods and Packages

15.4.1 Useful Packages

In this section, we explore Python's extensive collection of packages, offering tailored solutions for diverse tasks. Python's packages are specialized tools, empowering developers to address specific challenges efficiently. From data manipulation to machine learning, web development to scientific computing, Python packages provide a robust foundation for diverse projects.

15.4.1.1 collections

Python's **collections** module provides a **Counter** class, which is a specialized dictionary designed for counting hashable objects. One lesser-known feature of Counter is that it supports arithmetic operations like **addition**, **subtraction**, **intersection**, and **union**. This is particularly useful when you're dealing with counting occurrences of items across different datasets or need to perform set-like operations on the counts themselves.

```python
from collections import Counter

# Define two Counters
counter1 = Counter({'a': 3, 'b': 1, 'c': 2})
counter2 = Counter({'a': 1, 'b': 2, 'd': 1})

# Addition: Adds counts from two counters
print("Addition:", counter1 + counter2)
# Prints "Addition: Counter({'a': 4, 'b': 3, 'c': 2, 'd': 1})"

# Subtraction: Subtracts counts, but keeps only positive results
print("Subtraction:", counter1 - counter2)
# Prints "Subtraction: Counter({'a': 2, 'c': 2})"

# Intersection: Keeps only positive counts common to both counters
print("Intersection:", counter1 & counter2)
# Prints "Intersection: Counter({'a': 1, 'b': 1})"

# Union: Keeps maximum counts from both counters
print("Union:", counter1 | counter2)
# Prints "Union: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 1})"
```

Python's **collections** module provides a class called **defaultdict**, which is a subclass of the built-in dictionary (dict) class. The **defaultdict** is similar to a regular dictionary, but it allows you to specify a default value factory for missing keys. This means that when you access a key that doesn't exist, instead of raising a **KeyError**, defaultdict will create the key and assign it a default value returned by the factory function.

```python
from collections import defaultdict

# Create a defaultdict with int as the default value factory
d = defaultdict(int)

# Accessing a non-existent key will create it with a default value of 0
print(d['a'])  # Output: 0

# You can also specify a different default value factory
d = defaultdict(list)

# Accessing a non-existent key will create it with an empty list as the default value
print(d['b'])  # Output: []

# You can use any callable as the default value factory
d = defaultdict(lambda: 'default')

# Accessing a non-existent key will create it with 'default' as the default value
print(d['c'])  # Output: 'default'
```

15.4.1.2 itertools

Python's **itertools** module provides functions called **combinations** and **combinations_with_replacement**, which generates all possible **combinations** of a given length from the elements of an **iterable**, including combinations with repeated elements.

```python
from itertools import combinations

# Generate combinations of length 2 from the elements 'A', 'B', 'C' without replacement
combinations = combinations(['A', 'B', 'C'], 2)

```

```
# Print the generated combinations
for combo in combinations:
    print(combo, end=',')
# Prints "('A', 'B'),('A', 'C'),('B', 'C'),"
```

```
from itertools import combinations_with_replacement

# Generate combinations of length 2 from the elements 'A', 'B', 'C' with replacement
combinations = combinations_with_replacement(['A', 'B', 'C'], 2)

# Print the generated combinations
for combo in combinations:
    print(combo, end=',')
# Prints "('A', 'A'),('A', 'B'),('A', 'C'),('B', 'B'),('B', 'C'),('C', 'C'),"
```

15.4.1.3 slice

Python has a built-in **slice** object that can be used to slice **sequences** like **lists**, **tuples**, and **strings**. While slicing with regular slicing syntax (list[start:end:step]) is common, creating and using slice objects directly can be powerful, especially when working with multiple slices or when you want to reuse the same slice configuration.

```
# Create a slice object with start, stop, and step
my_slice = slice(1, 5, 2)

# Apply the slice to a list
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
sliced_result = my_list[my_slice]

print(sliced_result)  # Output: ['b', 'd']

# You can also use the slice object with other sequences like strings
my_string = "Hello, World!"
sliced_string = my_string[my_slice]

print(sliced_string)  # Output: 'el'
```

In this example, slice(1, 8, 2) creates a **slice** object that starts at index 1 ('b'), ends at index 8 ('i'), and steps by 2. So, applying this slice to the list ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'], it selects elements at indices 1, 3, 5, and 7, which correspond to the values 'b', 'd', 'f', and 'h', respectively.

```
# Create a slice object with start, stop, and step
my_slice = slice(1, 8, 2)

# Apply the slice to a list
my_list = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
sliced_result = my_list[my_slice]

print(sliced_result)  # Output: ['b', 'd', 'f', 'h']
```

15.4.1.4 functools

Python's **functools** module provides a function called **singledispatch**, which allows you to create a single-dispatch generic function. It allows you to define a function behavior based on the type of the first argument. This is particularly useful for creating functions that can handle different types of input in a flexible and extensible way, similar to **method overloading** in other programming languages.

```python
from functools import singledispatch

@singledispatch
def my_func(arg):
    print("Default implementation:", arg)

@my_func.register(int)
def _(arg):
    print("Processing an integer:", arg)

@my_func.register(str)
def _(arg):
    print("Processing a string:", arg)

@my_func.register(list)
def _(arg):
    print("Processing a list:", arg)

# Test the function with different types of input
my_func(10)          # Output: Processing an integer: 10
my_func("Hello")     # Output: Processing a string: Hello
my_func([1, 2, 3])   # Output: Processing a list: [1, 2, 3]
```

Python's **functools** module provides a decorator called **lru_cache**, which stands for "Least Recently Used Cache." It's a built-in memoization decorator that caches the results of a function and reuses them when the same inputs occur again. **lru_cache** Uses a least-recently-used (LRU) eviction policy. This means that when the cache reaches its maximum size (specified by the maxsize argument), the least recently used entries are discarded to make room for new ones.

```python
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

Python's **functools** module provides a function called **cache**, which allows you to cache the results of a function call and reuse them when the same inputs occur again. This can significantly improve the performance of your code, especially when dealing with expensive computations or I/O operations. This was introduced in python 3.9. Unlike **lru_cache**, **cache** Does not have a built-in eviction policy. It simply caches results until the program exits or until the function is explicitly uncached. Because it never needs to evict old values, this is smaller and faster than **lru_cache** with a size limit.

```python
from functools import cache

@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)       # just looks up cached value result
120
```

```
>>> factorial(12)      # makes two new recursive calls, the other 10 are cached
479001600
```

15.4.1.5 typing

Python's **typing** module provides a way to specify type hints for variables, function arguments, and return values. While type hints are often used for improving code readability and catching errors early during development, you can also use them for **runtime** type checking using the typing module's **runtime_checkable** decorator along with the **Protocol** class. In this example, Printable is defined as a protocol using the Protocol class. It specifies a method print that takes no arguments and returns None. By decorating Printable with @runtime_checkable, you enable runtime type checking for objects that claim to conform to the protocol. The print_if_possible function takes an argument of type Printable and calls its print method. When called with an object that conforms to the Printable protocol (like Printer), it works fine. However, if called with an object that does not conform to the Printable protocol (like NotPrinter), it raises a TypeError, allowing you to catch potential type errors at runtime.

```
from typing import Protocol, runtime_checkable

# Define a protocol with type hints for methods
@runtime_checkable
class Printable(Protocol):
    def print(self) -> None:
        pass

# A class that conforms to the Printable protocol
class Printer:
    def print(self) -> None:
        print("Printing...")

# A class that does not conform to the Printable protocol
class NotPrinter:
    def display(self) -> None:
        print("Displaying...")

def print_if_possible(obj: Printable) -> None:
    obj.print()

printer = Printer()
not_printer = NotPrinter()

# This will work fine since Printer conforms to the Printable protocol
print_if_possible(printer)

# This will raise a TypeError since NotPrinter does not conform to the Printable protocol
print_if_possible(not_printer)
```

15.4.1.6 enum

Python's **enum** module provides a powerful way to create **enumerations** in Python. Enumerations allow you to define symbolic names (members) for a set of unique values, which can make your code more readable and maintainable. One lesser-known feature of enumerations is the ability to create them dynamically using the Enum class's functional API. This allows you to create enumerations without explicitly subclassing Enum.

```
from enum import Enum

# Define a dynamic enumeration
DynamicEnum = Enum("DynamicEnum", ["VALUE1", "VALUE2", "VALUE3"])

```

```
# Access enumeration members
print(DynamicEnum.VALUE1)  # Output: DynamicEnum.VALUE1
print(DynamicEnum.VALUE2)  # Output: DynamicEnum.VALUE2
print(DynamicEnum.VALUE3)  # Output: DynamicEnum.VALUE3
```

To access an **enum** name from an **enum** class based on an the value, you can write a method to access the enum values and compare to another value.

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

def get_enum_from_value(value):
    for color in Color:
        if color.value == value:
            return color
    raise ValueError("No enum member with that value")

color = get_enum_from_value(2)
print(color)  # Output: Color.GREEN
```

15.4.1.7 contextlib

Python's **contextlib** module provides a decorator called **contextmanager**, which allows you to create context managers using generator functions. Context managers are objects that define __enter__ and __exit__ methods, and they are commonly used with the with statement to manage resources, such as opening and closing files or acquiring and releasing locks.

```
from contextlib import contextmanager

@contextmanager
def open_file(filename, mode):
    file = open(filename, mode)
    try:
        yield file
    finally:
        file.close()

# Usage of the context manager
with open_file("example.txt", "w") as f:
    f.write("Hello, context managers!")
```

Python's **contextlib** module provides utilities for working with context managers. One lesser-known feature of the **contextlib** module is the **redirect_stdout** and **redirect_stderr** context managers, which allow you to temporarily redirect the standard output and standard error streams to a specified file or file-like object.

```
from contextlib import redirect_stdout
import io

# Create a StringIO object to redirect output
output_buffer = io.StringIO()

# Redirect stdout to the StringIO object
with redirect_stdout(output_buffer):
    print("This will be captured by the redirect_stdout context manager")

```

```
# Get the captured output from the StringIO object
captured_output = output_buffer.getvalue()

print("Captured Output:", captured_output)
# Prints "Captured Output: This will be captured by the redirect_stdout context manager"
```

15.4.2 Useful Methods

Thsi section is a collection of useful methods that can be adapted or used for many applications. These are all methods that I've written or heavily modified for my own use and have found useful on multiple occasions.

15.4.2.1 Logging and Application Output

To perform **logging** for an application, you often want to create a **log file** named based on the application name and the current date. This can be expanded to be contained in a class and called during initialization or just used as standalone.

```
#!/bin/python3
import os
import sys

# Used for gathering time and date data.
from datetime import datetime


def set_log_file_name():
    """
    Returns the log file name. If it has not yet been set, creates the log filename that
        ↪ contains the date and then returns it.

    The desired behavior is that one log_file is used per run of the application.
    To account for day changes while the program is running, log_file must be created at
        ↪ application runtime.
    """
    global log_file

    # Get the script name with full path
    script_path = os.path.abspath(sys.argv[0])
    # Get the script name without the extension.
    script_name = os.path.basename(script_path).split('.py')[0]

    # Check if the log file already has been set.
    if log_file == "":
        today = datetime.today().strftime("%Y%m%d")
        log_file = f"{script_name}_log_{today}"

    return log_file

# The log file to be used throughout the file.
log_file = ""
set_log_file_name()
```

With a **log file** defined, you can perform **logging** by appending messages to a log file. The message can include a **time stamp** for referencing when various tasks were performed at a later date.

```
# Used for gathering time and date data.
from datetime import datetime

```

```

def log_text(text, include_time_stamp=True):
    """
    This will output text to the logfile with an optional time stamp.
    Args:
        text (str): The text to be logged.
        timestamp (bool): Option to include timestamp in logging.
    """
    # Set's the appropriate timestamp value to use.
    if include_time_stamp:
        timestamp_text = datetime.now().strftime("[%Y-%m-%d %H %M %S]")
    else:
        timestamp_text = ""

    # Logs the output. This assumes log_file is defined.
    with open(log_file, 'a') as log:
        log.write(f"{timestamp_text} {text}\n")
```

When writing an application with complex or varied output, it is often useful to **print** messages to the terminal using various **color codes**. To do this, you can write a method that takes either various types of options (each corresponding to a color code) or takes a color name directly. You can also modify it to optionally automatically log the information using the log_text() method defined above. This is a highly adaptable code snippet.

```
def output_text(text:str, option:str=None , color:str=None , log=True):
    """
    Print text in different colors based on the provided option. This will also log the
        ↪ output text (including a time stamp).

    Args:
        text (str): The text to be printed and logged.
        option (str): The color option for the text. Valid options are defined in the
            ↪ color mappings list (second element). Default = None.
        color (str): The color name for the text. Valid options are defined in the color
            ↪ mappings list (first element). Default = None.
        log (bool): Determines whether to log the text or not. Default = True.

    Returns:
        None

    Note:
        This function uses ANSI escape codes for color formatting. Colors may not be
            ↪ displayed correctly in all environments. The color and option parameters
            ↪ should not both be used. If they are, the color will override the option
            ↪ parameter.
    """
    # Defines the default behaviour.
    if color == None and option == None:
        color = "white"
    if color != None and option != None:
        option = None
    if color == None:
        color = ""
    if option == None:
        option = ""

    # Define the list of color mappings
    color_mappings = [
        ("white",   "text",    "\033[0m"),      # White/default/reset
        ("red",     "error",   "\033[91m"),     # Red
```

```
        ("green",   "success", "\033[92m"),     # Green
        ("yellow",  "warning", "\033[93m"),     # Yellow
        ("blue",    "note",    "\033[94m"),     # Blue
    ]

    # Set's the reset value for normal text.
    reset_code = "\033[0m"
    found_match = False

    for color_name, color_type, color_code in color_mappings:
        if color.lower() == color_name:
            print(f"{color_code}{text}{reset_code}")
            found_match = True
            break
        elif option.lower()  == color_type:
            print(f"{color_code}{text}{reset_code}")
            found_match = True
            break

    # If no color match was found, print the text.
    if not found_match:
            print(text)

    if log:
        log_text(text)
```

15.1: ANSI Color Codes For Terminal Output

The American National Standards Institute (**ANSI**) **color codes** in the above can be expanded further if the application needs more color outputs. The below chart can be used.

Color Name	ANSI Escape Code	Example
White	\033[0m	
Black	\033[30m	Text
Red	\033[91m	Text
Green	\033[92m	Text
Yellow	\033[93m	Text
Blue	\033[94m	Text
Magenta	\033[35m	Text
Cyan	\033[36m	Text
Gray	\033[90m	Text
Purple	\033[95m	Text
Orange	\033[33m	Text

15.4.2.2 Config Files

For a basic and somewhat generic **config file framework**, you can use the following setup. The config variables and default values are stored in a map. This method will then update any variable/value pairs that are found in a configuration file. The config file should be setup so that each line contains a 'variable=value' pair. The value's must match the type that are defined in the configuration variables map. Lines that start with a '#' are considered comment lines and ignored. The values can then all be called by calling the configuration map.

```
# Example config file setup for the below framework.
name = John
age = 30
```

```
weight = 70.5
```

```
# Example config_variables
config_vars = {
    'name':           'default_name',
    'age':            0,
    'weight':         0.0,
    'is_student':     False,
    'favorite_color': 'red'
}

def load_config_file(config_file, config_variables, verbose=False):
    """
    Load configuration variables from a file and update their values.

    Args:
        config_file (str): The path to the configuration file.
        config_variables (map): The map of configuration variables.
            Should be formatted with default values for each possible configuration
                ↪ variable desired:
            {'name':'default', 'age':0, 'weight':0.0, ...}
        verbose (bool, optional): If True, print verbose messages indicating the
            ↪ configuration variables being set.
                                  Defaults to False.

    Returns:
        None

    Raises:
        FileNotFoundError: If the specified config_file is not found.

    Notes:
        - Configuration variables are updated based on the values specified in the config
            ↪ file.
        - The config file should have lines in the format "variable = value".
        - Each variable's value will be converted to the specified type as per the
            ↪ config_variables dictionary.
    """
    try:
        with open(config_file, 'r') as file:
            for line in file:
                # Skip commented lines.
                if line.startswith('#'):
                    continue

                if '=' in line:
                    variable, value = map(str.strip, line.strip().split('='))
                    if variable in config_variables:
                        # Retrieve default value and type
                        value_type = type(config_variables[variable])
                        # Convert the value to the proper type
                        converted_value = value_type(value)
                        config_variables[variable] = converted_value
                        if verbose:
                            print(f"Setting config variable: {variable} -> {
                                ↪ converted_value}")
                    elif verbose:
                        print(f"WARNING: Invalid variable detected in config file: {
                            ↪ variable}")
```

```
52      except FileNotFoundError:
53          print("ERROR: Config file not found.")
```

References

[1] http://www.cplusplus.com/doc/tutorial/pointers/

[2] http://www.cplusplus.com/reference/chrono/

[3] https://os.mbed.com/handbook/C-Data-Types

[4] Kumar, Chandan. “10 Useful Linux Networking Commands.” Geek Flare, 11 Feb. 2018, geekflare.com/linux-networking-commands/.

[5] Parker, Steve. “Shell Scripting Tutorial.” The Shell Scripting Tutorial, www.shellscript.sh/.

[6] https://git-scm.com/docs/

[7] https://www.toptal.com/git/the-advanced-git-guide

Index