Mawhiba-Oxmedica
Universal Enrichment Program
29 June - 18 July 2024

موهبة

Course Reader
for
Cybersecurity & Cryptography

Tutor: Mr. Omar Choudhry

Student's Name:

Table of Contents

1 Course Introduction *5*

2 Career Paths in Cybersecurity *5*

2.1 What is a cybersecurity career? **5**

2.2 How much can you make? **6**

2.3 Skills needed for a cybersecurity career **6**

3 Critical Cybersecurity Studies *9*

4 Programming *9*

4.1 What is Programming? **10**

4.2 Key Concepts in Programming **10**

4.3 Popular Programming Languages **10**

4.4 Tools and Environment **11**

4.5 Cheat Sheets **11**

5 Version Control *19*

5.1 What is Version Control? **19**

5.2 Git: The Leading Version Control System **19**

5.3 GitHub: A Hub for Collaborative Projects **20**

6 Architecture of Computers *21*

6.1 Definitions **21**

6.2 Boolean Logic **22**

6.2.1 Truth Tables 22

6.2.2 Logic Gates 22

6.3 Number Systems **23**

6.4 Memory **23**

6.5 Operating Systems **24**

6.6 CPUs and GPUs **24**

6.7 Assembly Code **25**

6.8 Buffer Overflow **25**

7 Database Systems *26*

7.1 Definitions **26**

7.2 Boolean Logic **27**

7.3 Database Architecture **29**

7.3.1 Relational Algebra 29

7.3.2 ER Modelling 30

7.3.3 Normalisation and Database Design 30

7.3.4 Big Data and Database Management Systems 30

7.3.5 Maintaining Data Privacy Regulations 31
7.3.6 Visualising Databases 31
7.4 SQL Injections and Vulnerabilities 31
8 Cryptographic Techniques 32
8.2 Overview of Encryption and Decryption Process 32
8.2.1 Public and Private Cryptography 33
8.2.2 Performing Decryption 33
8.2.3 Mathematical Encryption and Decryption 33
8.3 Issues in Cryptographic Techniques 33
8.4 Advanced Cryptographic Techniques 34
9 Network Security Concepts 34
9.1 TCP/IP Networking Architecture 34
9.2 Common Networking Threats 35
9.3 Network Defences 35
9.3.2 Firewalls 36
9.3.3 Packet Filtering 37
9.2.4 Defence Against Man-in-the-Middle Attacks 37
9.2.5 Domain Name Systems (DNS) 38
9.3 Network Monitoring Tools 39
9.3.1 Wireshark 39
9.3.2 Cisco Packet Tracer 40
10 Cybersecurity Legislation and Regulation 40
10.1 Saudi Arabia 41
10.2 United Kingdom 41
10.3 United States 41
10.4 European Union 41
11 Hacking and Cyber Attacks 42
11.1 Cyber Attacks 42
11.1.1 Brute Force Attacks 42
11.1.2 DDoS Attacks 42
11.1.3 Other Attacks 43
11.2 Malware and Viruses 43
11.2.1 Logic Bombs 43
11.2.2 Backdoors 44
11.2.3 Viruses 44
11.2.4 Supply Chain Attacks 45
11.2.5 Trojan Horses 45
11.2.6 Worms 46
11.2.7 Methods to Prevent Malware and Viruses 46
11.2.8 Input Validation 46
11.3 Threat-Vectors and Threat Agent 46
11.3.1 Email and Phishing Attacks 46
11.3.2 Web-Based Threats 47
11.3.3 Malicious Insider Threats 47

11.3.4 Removable Media 47
11.3.5 Wireless Networks 47
11.3.6 Supply Chain Attacks 48

Cover Page Photo: https://www.eenewseurope.com/en/automated-test-system-to-strengthen-post-quantum-cryptography/

1 Course Introduction

For all relevant information, see the pre-student handout. I have provided a link and QR code below, which contain extra resources for all sections of this course reader in a Google Docs document. This may be referred to later as the "*CC Doc*".

https://bit.ly/OxmedicaCC

One helpful resource is a YouTube playlist I created (the link is right at the top of the document) containing a list of YouTube videos related to this field. Some are technical and will teach you concepts, while others are story-based and will expose you to this vast field's history.

Remember that this document is not for you to learn the content of this course. It is supplementary material to the main content taught in lectures and through practical activities. You are not expected to understand and know every single definition in this document. The goal was to briefly describe everything without going into too much detail. Feel free to ask any questions.

2 Career Paths in Cybersecurity

Taken from Coursera articles. See CC Doc for more.

Pursuing a career in cybersecurity means joining a booming industry where available jobs outnumber qualified candidates. According to the US Bureau of Labor Statistics (BLS), the number of cybersecurity jobs is expected to increase by 32 percent between 2022 and 2032. The COVID-19 pandemic has only accelerated this demand.

2.1 What is a cybersecurity career?

A cybersecurity career involves working in roles focused on protecting technological assets from unauthorised or criminal use while ensuring information remains accessible and confidential when appropriate. As a cybersecurity professional, you will demonstrate an understanding of how computers and other business technology work.

In a cybersecurity career, you could play a part in several areas:

- Keep all business IT current (with regular updates and software and system patches)
- Educate employees about cyber threats (e.g., social engineering, weak passwords)
- Monitor network traffic to identify potential threats.
- Block unauthorised access.
- Develop response plans in case of an attack.
- Maintain compliance with industry standards.
- Ensure compliance with government regulations.
- Create security roles.
- Install authentication and access control.
- Evaluate new products.
- Test protections and business recovery plans.

2.2 How much can you make?

Cybersecurity professionals get paid well for their skills, even at the entry-level level. As you gain experience and move into more advanced roles, salaries also often increase. To give you an idea of what's possible, here's a look at the average total pay of several cybersecurity jobs in the US in October 2023, according to Glassdoor.

- Intrusion detection specialist: $71,102
- Junior cybersecurity analyst: $91,286
- Digital forensic examiner: $119,322
- IT security administrator: $87,805
- Incident response analyst: $67,877
- Cybersecurity consultant: $105,435
- Information security analyst: $98,497
- Ethical hacker: $133,458
- Penetration tester: $100,579
- Security engineer: $114,898
- Cybersecurity manager: $160,020
- Security architect: $211,207
- Chief information security officer: $301,873

2.3 Skills needed for a cybersecurity career

Technical and workplace skills are essential to succeeding in your cybersecurity career. Employers seek the highest-quality candidates, so it can be helpful to develop these skills.

Technical

To launch your cybersecurity career, you'll typically need to have at least some of the following technical skills:

- System, software, hardware installation and maintenance
- Web application, endpoint, and cloud security
- Computer programming and scripting
- Disk management
- Security information and event management (SIEM)
- Security orchestration, automation, and response (SOAR)
- Knowledge of security audit practices
- Understanding of antivirus and malware protections
- Digital forensics
- Cybersecurity framework knowledge

Workplace

- Develop your career in the cybersecurity industry with the following additional workplace skills:
- Communication
- Leadership
- Logical reasoning and creative problem-solving
- Ability to work alone but also collaborate with others
- Attention to detail
- Desire and willingness to engage in ongoing, continual learning

Figure 1

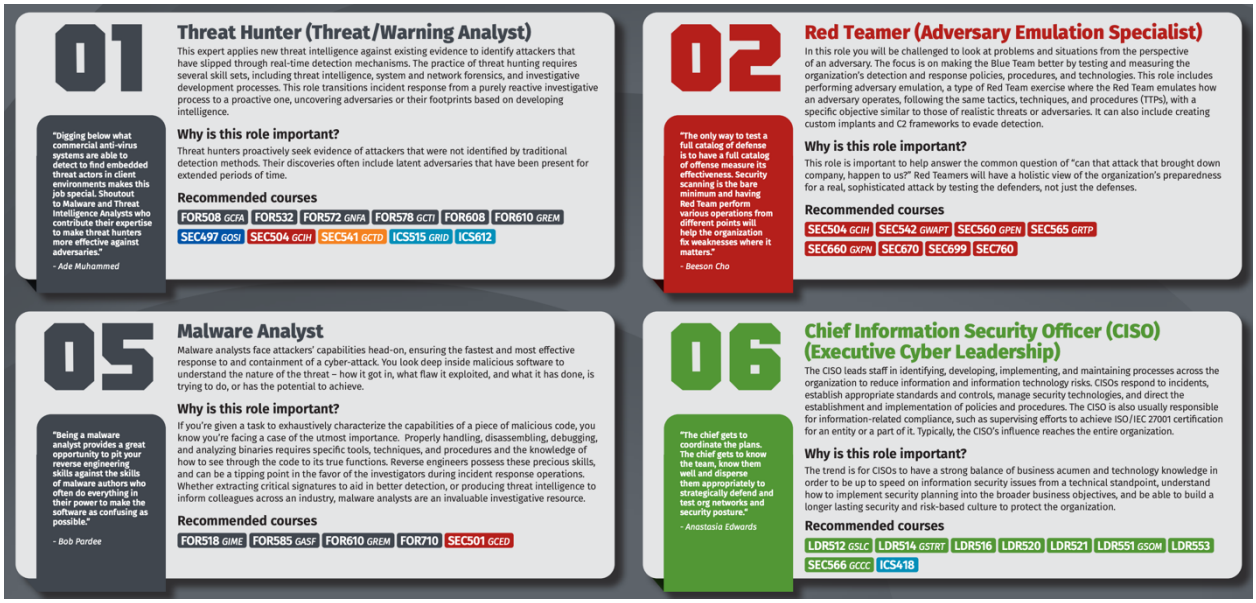


Figure 2

[1] https://www.ncsc.gov.uk/files/NCSC%2010%20Steps%20To%20Cyber%20Security%20NCSC.pdf
[2] https://www.sans.org/cybersecurity-careers/20-coolest-cyber-security-careers/

03 Digital Forensic (Cyber Defense Forensics Analyst)

This expert applies digital forensic skills to a plethora of media that encompass an investigation. The practice of being a digital forensic examiner requires several skill sets, including evidence collection, computer, smartphone, cloud, and network forensics, and an investigative mindset. These experts analyze compromised systems or digital media involved in an investigation that can be used to determine what really happened. Digital media contain footprints that physical forensic data and the crime scene may not include.

"Forensics is about diving deep into any system and device and locating the problem so as to develop a solution."
- Patricia M

"Data doesn't lie, and the digital forensic analyst looks at the data to convey the stories that they tell."
- Anthony Wo

Why is this role important?

You are the sleuth in the world of cybersecurity, searching computers, smartphones, cloud data, and networks for evidence in the wake of an incident/crime. The opportunity to learn never stops. Technology is always advancing, as is your career.

Recommended courses

FOR308 | FOR498 GBFA | FOR500 GCFE | FOR508 GCFA | FOR509 GCFR | FOR518 GIME | FOR532 | FOR572 GNFA | FOR585 GASF | SEC501 GCED

04 Purple Teamer

In this fairly recent job position, you have a keen understanding of both how cybersecurity defenses ("Blue Team") work and how adversaries operate ("Red Team"). During your day-to-day activities, you will organize and automate emulation of adversary techniques, highlight possible new log sources and use cases that help increase the detection coverage of the SOC, and propose security controls to improve resilience against the techniques. You will also work to help coordinate effective communication between traditional defensive and offensive roles.

"The combination of red team blue team operations is very interesting and you get to see both sides. I have been on a Purple Team for a while now and it has driven a lot of positive change for us."
- Andrew R

Why is this role important?

Help blue and red understand one another better! Blue Teams have traditionally been talking about security controls, log sources, use cases, etc. On the other side Red Teams traditionally talk about payloads, exploits, implants, etc. Help bridge the gap by ensuring red and blue are speaking a common language and can work together to improve the overall cybersecurity posture of the organization!

Recommended courses

SEC599 GDAT | SEC699 | SEC504 GCIH | SEC568 | SEC598

07 Blue Teamer – All-Around Defender (Cyber Defense Analyst)

This job, which may have varying titles depending on the organization, is often characterized by the breadth of tasks and knowledge required. The all-around defender and Blue Teamer is the person who may be a primary security contact for a small organization, and must deal with engineering and architecture, incident triage and response, security tool administration and more.

"In this day and age, we need guys that are good at defense and understand how to harden systems."
- David O

Why is this role important?

This job role is highly important as it often shows up in small to mid-size organizations that do not have budget for a full-fledged security team with dedicated roles for each function. The all-around defender isn't necessarily an official job title as it is the scope of the defense work such defenders may do – a little bit of everything for everyone.

Recommended courses

SEC450 | SEC503 GCIA | SEC505 GCWN | SEC511 GMON | SEC530 GDSA | SEC555 GCDA | SEC586

08 Security Architect (NICE) and Engineer

Design, implement, and tune an effective combination of network-centric and data-centric controls to balance prevention, detection, and response. Security architects and engineers are capable of looking at an enterprise defense holistically and building security at every layer. They can balance business and technical requirements along with various security policies and procedures to implement defensible security architectures.

"A security architect needs to understand work flows, networks, business requirements, project plans and sometimes even budget restraints. A very diversified role!"
- Chris Bodill

Why is this role important?

A security architect and engineer is a versatile Blue Teamer and cyber defender who possesses an arsenal of skills to protect an organization's critical data, from the endpoint to the cloud, across networks and applications.

Recommended courses

SEC503 GCIA | SEC505 GCWN | SEC511 GMON | SEC530 GDSA | SEC549

11 OSINT Investigator/Analyst

These resourceful professionals gather requirements from their customers and then, using open sources and mostly resources on the internet, collect data relevant to their investigation. They may research domains and IP addresses, businesses, people, issues, financial transactions, and other targets in their work. Their goals are to gather, analyze, and report their objective findings to their clients so that the clients might gain insight on a topic or issue prior to acting.

"Being an OSINT investigator allows me to extract information in unique and clever ways and I am never bored. One day I'm working on a fraud investigation and the next I'm trying to locate a missing person. This job always tests my capabilities, stretches my critical thinking skills, and lets me feel like I'm making a difference."
- Rebecca Ford

Why is this role important?

There is a massive amount of data that is accessible on the internet. The issue that many people have is that they do not understand how best to discover and harvest this data. OSINT investigators have the skills and resources to discover and obtain data from sources around the world. They support people in other areas of cybersecurity, intelligence, military, and business. They are the finders of things and the knowers of secrets.

Recommended courses

SEC497 GOSI | SEC587 | FOR578 GCTI

12 Technical Director (Information Systems Security Manager)

This expert defines the technological strategies in conjunction with development teams, assesses risk, establishes standards and procedures to measure progress, and participates in the creation and development of a strong team.

"A technical director must have strong cybersecurity knowledge, a strategic view of the organization's infrastructure and what's to come, and communication skills. These things are hard to get, and I would imagine this job to be very challenging, no matter the organization size or business."
- Francisco Lugo

Why is this role important?

With a wide range of technologies in use that require more time and knowledge to manage, a global shortage of cybersecurity talent, an unprecedented migration to cloud, and legal and regulatory compliance often increasing and complicating the matter more, a technical director plays a key role in successful operations of an organization.

Recommended courses

LDR512 GSLC | LDR514 GSTRT | LDR516 | LDR551 GSOM | SEC566 GCCC | ICS418

09 Cyber Defense Incident Responder/Law Enforcement Counterintelligence Forensics Analyst

This dynamic and fast-paced role involves identifying, mitigating, and eradicating attackers while their operations are still unfolding.

"Incidents are bound to occur and it is important that we have people with the right skill set to manage and mitigate the loss to the organization from these incidents."
- Anita Ali

Why is this role important?

While preventing breaches is always the ultimate goal, one unwavering information security reality is that we must assume a sufficiently dedicated attacker will eventually be successful. Once it has been determined that a breach has occurred, incident responders are called into action to locate the attackers, minimize their ability to damage the victim, and ultimately remove them from the environment. This role requires quick thinking, solid technical and documentation skills, and the ability to adapt to attacker methodologies. Further, incident responders work as part of a team, with a wide variety of specializations. Ultimately, they must effectively convey their findings to audiences ranging from deep technical to executive management.

Recommended courses

FOR508 GCFA | FOR509 GCFR | FOR518 GIME | FOR532 | FOR572 GNFA | FOR578 GCTI | FOR608 | FOR610 GREM | FOR710 | SEC402 | ICS515 GRID | SEC504 GCIH

10 Cybersecurity Analyst/Engineer (Systems Security Analyst)

As this is one of the highest-paid jobs in the field, the skills required to master the responsibilities involved are advanced. You must be highly competent in threat detection, threat analysis, and threat protection. This is a vital role in preserving the security and integrity of an organization's data.

"It doesn't become much more versatile than in this role, as oftentimes you'll be challenged with whathever tasks or projects customers or managers envision, ranging from simple analysis support to introducing new solutions and implementing whole services such as a SOC."
- Harun Kuessner

Why is this role important?

This is a proactive role, creating contingency plans that the company will implement in case of a successful attack. Since cyber attackers are constantly using new tools and strategies, cybersecurity analysts/engineers must stay informed about the tools and techniques out there to mount a strong defense.

Recommended courses

SEC401 GSEC | SEC450 | SEC501 GCED | SEC503 GCIA | SEC530 GDSA | SEC555 GCDA | SEC504 GCIH | SEC554 | FOR508 GCFA | FOR509 GCFR | LDR551 GSOM | SEC510 GPCS | SEC540 GCSA | SEC549 | ICS410 GICSP | ICS456 GCIP

13 Cloud Security Analyst

The cloud security analyst is responsible for cloud security and day-to-day operations. This role contributes to the design, integration, and testing of tools for security management, recommends configuration improvements, assesses the overall cloud security posture of the organization, and provides technical expertise for organizational decision-making.

"This role is essential to find and patch vulnerabilities in the cloud environment to ensure that crackers and hackers are unauthorized in cloud environments."
- Ben Yee

Why is this role important?

With an unprecedented move from traditional on-premise solutions to the cloud, and a shortage of cloud security experts, this position helps an organization position itself thoughtfully and securely in a multicloud environment necessary for today's business world.

Recommended courses

SEC488 GCLD | SEC510 GPCS | SEC541 GCTD | SEC401 GSEC | FOR509 GCFR | SEC588 GCPN

14 Intrusion Detection/SOC Analyst (Cyber Defense Analyst)

Security Operations Center (SOC) analysts work alongside security engineers and SOC managers to implement prevention, detection, monitoring, and active response. Working closely with incident response teams, a SOC analyst will address security issues when detected, quickly and effectively. With an eye for detail and anomalies, these analysts see things most others miss.

"The intrusion analyst is the guard at the gate and can get great job satisfaction from detecting and stopping network intrusions."
- Chuck Ballard

Why is this role important?

SOC analysts help organizations have greater speed in identifying attacks and remedying them before they cause more damage. They also help meet regulation requirements that require security monitoring, vulnerability management, or an incident response function.

Recommended courses

SEC450 | SEC503 GCIA | SEC511 GMON | SEC555 GCDA | FOR508 GCFA | FOR572 GNFA | FOR532 | SEC504 GCIH

17 Application Pen Tester (Secure Software Accessor)

Application penetration testers probe the security integrity of a company's applications and defenses by evaluating the attack surface of all in-scope vulnerable web-based services, client-side applications, servers-side processes, and more. Mimicking a malicious attacker, app pen testers work to bypass security barriers in order to gain access to sensitive information or enter a company's internal systems through techniques such as pivoting or lateral movement.

"It is not only about using existing tools and methods, you must be creative and understand the logic of the application and make guesses about the infrastructure."
- Dan-Mihai Negrea

Why is this role important?

Web applications are critical for conducting business operations, both internally and externally. These applications often use open source plugins which can put these apps at risk of a security breach.

Recommended courses

SEC542 GWAPT | SEC560 GPEN | SEC575 GMOB | SEC588 GCPN | SEC660 GXPN | SEC760

18 ICS/OT Security Assessment Consultant (ICS/SCADA Security Engineer)

One foot in the exciting world of offensive operations and the other foot in the critical process control environments essential to life. Discover system vulnerabilities and work with asset owners and operators to mitigate discoveries and prevent exploitation from adversaries.

"Working in this type of industry, I can see how the demand is increasing so rapidly that companies starting to desperately looking for people with proper skillsets."
- Ali Alhajhouj

Why is this role important?

Security incidents, both intentional and accidental in nature, that affect OT (primarily in ICS systems) can be considered to be high-impact but low-frequency (HILF); they don't happen often, but when they do the cost to the business can be considerable.

Recommended courses

ICS410 GICSP | ICS456 GCIP | ICS515 GRID | ICS612 | SEC560 GPEN

3 Critical Cybersecurity Studies

Today, hackers are growing more sophisticated and security breaches are becoming more frequent. Protecting against risks and threats to an organisation's data is necessary. The Cybersecurity and Infrastructure Security Agency (CISA) reports that 47 percent of adults in the US have had their personal data exposed by cyber criminals. If you consider that a data breach can cost an average of $9.44 million in the US, it's clear why there is a high demand for cybersecurity.

Cybersecurity protects an organisation's computer systems, networks, and data from unauthorised access or damage. Cyberattacks and threats can access, change, or destroy sensitive information such as passwords, financial information, medical records, and other confidential data.

Keeping current with the latest advances in this field is a good idea. You can see a selection of news here:

- https://cybersecuritynews.com/weekly-cyber-security-news-round-up/
- https://www.darkreading.com/
- https://www.wired.com/category/security/

See the CC Doc for a list of YouTube videos, images, articles, and case studies. In the course, we will look at a few specific examples.

4 Programming

Programming is a fundamental skill in the digital age, crucial for creating software, solving problems, and understanding how computers operate. This report provides an introductory overview of programming tailored for high school students, covering basic concepts, languages, tools, and applications.

4.1 What is Programming?

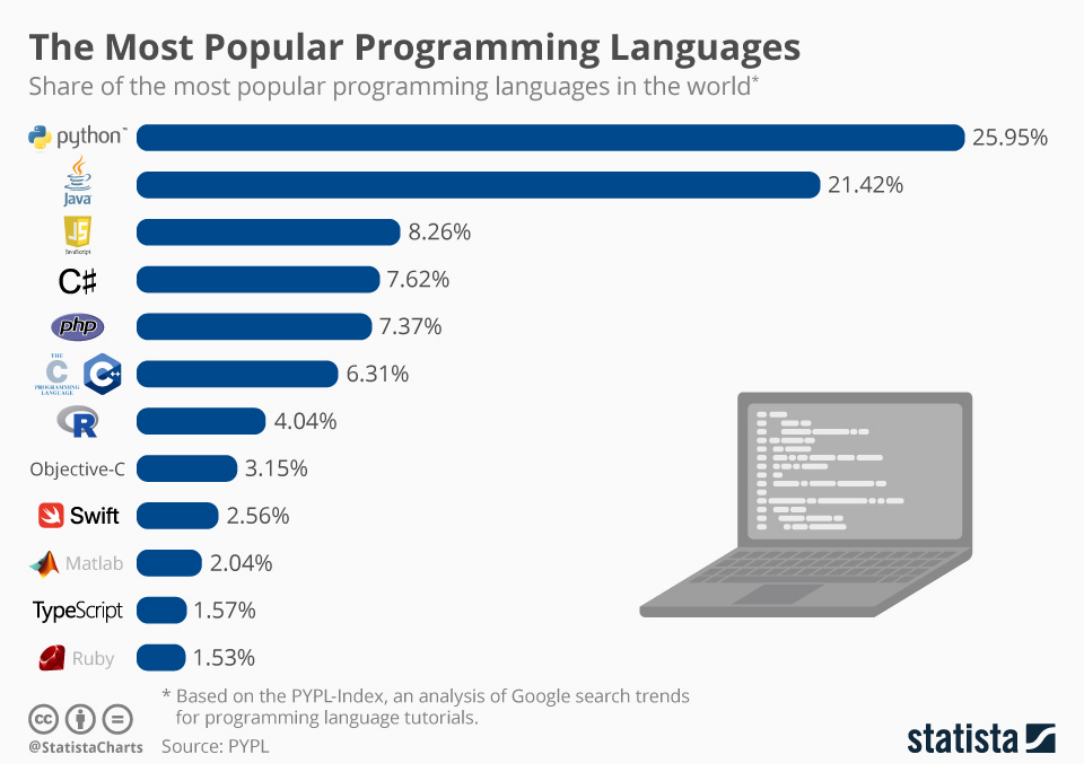
Programming, often called coding, is designing, and building an executable computer program to accomplish a specific computing result or perform a particular task. It involves tasks such as analysis, generating algorithms, profiling algorithms' accuracy and resource consumption, and implementing algorithms in a chosen programming language (commonly referred to as coding).

4.2 Key Concepts in Programming

- **Algorithms**: A step-by-step procedure or formula for solving a problem.
- **Variables**: Basic storage that can hold data that may change during the program's run. **Data Types** are the classification of data that tells the compiler or interpreter how the programmer intends to use the data. Common types include integers, floating-point numbers, and strings.
- **Control Structures**: Instructions that determine the flow of control in a program. These include decisions (if statements), loops (for, while), and branching (switch statements).
- **Functions** are blocks of code packaged as units that perform a specific task. They can be called when needed in the program.

4.3 Popular Programming Languages

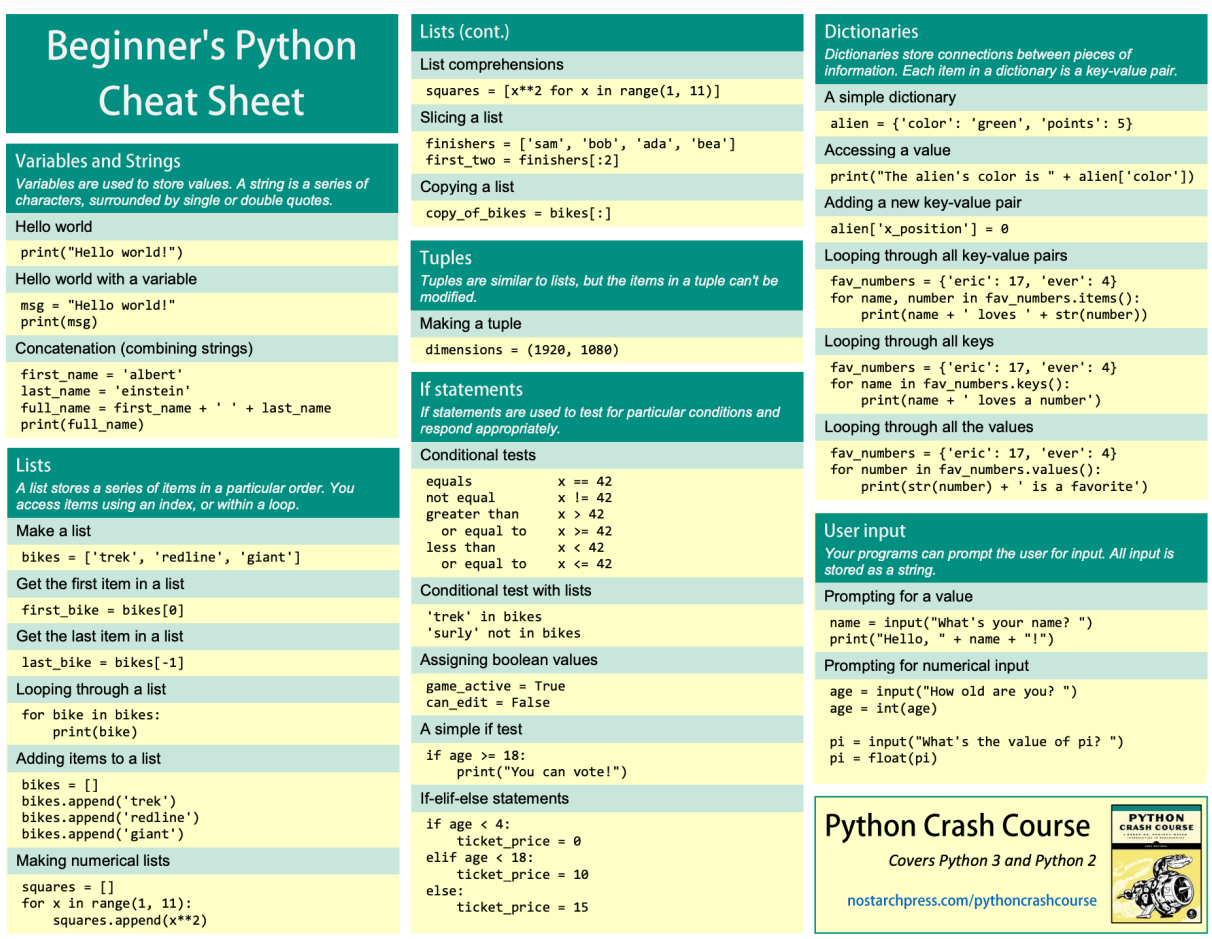
- **Python**: Known for its readability and simplicity, it is perfect for beginners.
- **Java**: Widely used in enterprise environments, known for its portability across platforms.
- **C++** is an extension of 'C' with enhanced features, commonly used for system/software development and games.
- **JavaScript**: Essential for web development, it runs in the browser and adds interactivity to web pages.



4.4 Tools and Environment

- **Integrated Development Environments (IDEs)**: Software applications that provide comprehensive facilities to programmers for software development. Examples include Visual Studio Code, PyCharm, and Eclipse.
- **Compilers and Interpreters**: Tools that transform the code written in a programming language into a form a computer can execute. Python uses an interpreter, while C++ uses a compiler.

4.5 Cheat Sheets



Beginner's Python Cheat Sheet

Variables and Strings
Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world
```
print("Hello world!")
```
Hello world with a variable
```
msg = "Hello world!"
print(msg)
```
Concatenation (combining strings)
```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists
A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list
```
bikes = ['trek', 'redline', 'giant']
```
Get the first item in a list
```
first_bike = bikes[0]
```
Get the last item in a list
```
last_bike = bikes[-1]
```
Looping through a list
```
for bike in bikes:
    print(bike)
```
Adding items to a list
```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```
Making numerical lists
```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions
```
squares = [x**2 for x in range(1, 11)]
```
Slicing a list
```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```
Copying a list
```
copy_of_bikes = bikes[:]
```

Tuples
Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple
```
dimensions = (1920, 1080)
```

If statements
If statements are used to test for particular conditions and respond appropriately.

Conditional tests
```
equals            x == 42
not equal         x != 42
greater than      x > 42
  or equal to     x >= 42
less than         x < 42
  or equal to     x <= 42
```
Conditional test with lists
```
'trek' in bikes
'surly' not in bikes
```
Assigning boolean values
```
game_active = True
can_edit = False
```
A simple if test
```
if age >= 18:
    print("You can vote!")
```
If-elif-else statements
```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries
Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary
```
alien = {'color': 'green', 'points': 5}
```
Accessing a value
```
print("The alien's color is " + alien['color'])
```
Adding a new key-value pair
```
alien['x_position'] = 0
```
Looping through all key-value pairs
```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```
Looping through all keys
```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```
Looping through all the values
```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

User input
Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value
```
name = input("What's your name? ")
print("Hello, " + name + "!")
```
Prompting for numerical input
```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

Python Crash Course
Covers Python 3 and Python 2
nostarchpress.com/pythoncrashcourse

Figure 3

[3] https://nostarch.com/

While loops

A while loop repeats a block of code as long as a certain condition is true.

A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")

make_pizza()
make_pizza('pepperoni')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Lists

What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

Changing an element

```
users[0] = 'valerie'
users[-2] = 'ronald'
```

Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []
users.append('val')
users.append('bob')
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

Pop the last item from a list

```
most_recent_user = users.pop()
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)
print(first_user)
```

List length

The len() function returns the number of items in a list.

Find the length of a list

```
num_users = len(users)
print("We have " + str(num_users) + " users.")
```

Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

Printing all items in a list

```
for user in users:
    print(user)
```

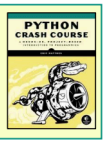
Printing a message for each item, and a separate message afterwards

```
for user in users:
    print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!")
```


Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



While loops

A while loop repeats a block of code as long as a certain condition is true.

A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")

make_pizza()
make_pizza('pepperoni')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Dictionaries

What are dictionaries?

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.

Defining a dictionary

Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.

Making a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
```

Accessing values

To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.

You can also use the get() method, which returns None instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.

Getting the value associated with a key

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

Getting the value with get()

```
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points', 0)

print(alien_color)
print(alien_points)
```

Adding new key-value pairs

You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.

This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.

Adding a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}

alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

Adding to an empty dictionary

```
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
```

Modifying values

You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.

Modifying values in a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

# Change the alien's color and point value.
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

Removing key-value pairs

You can remove any key-value pair you want from a dictionary. To do so use the del keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.

Deleting a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

Visualizing dictionaries

Try running some of these examples on pythontutor.com.

Looping through a dictionary

You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.

A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.

Looping through all key-value pairs

```
# Store people's favorite languages.
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

Looping through all the keys

```
# Show everyone who's taken the survey.
for name in fav_languages.keys():
    print(name)
```

Looping through all the values

```
# Show all the languages that have been chosen.
for language in fav_languages.values():
    print(language)
```

Looping through all the keys in order

```
# Show each person's favorite language,
#   in order by the person's name.
for name in sorted(fav_languages.keys()):
    print(name + ": " + language)
```

Dictionary length

You can find the number of key-value pairs in a dictionary.

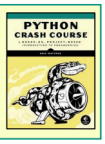
Finding a dictionary's length

```
num_responses = len(fav_languages)
```


Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Nesting — A list of dictionaries

It's sometimes useful to store a set of dictionaries in a list; this is called nesting.

Storing dictionaries in a list

```
# Start with an empty list.
users = []

# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
    }
users.append(new_user)

# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
    }
users.append(new_user)

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

You can also define a list of dictionaries directly, without using append():

```
# Define a list of users, where each user
#   is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

Nesting — Lists in a dictionary

Storing a list inside a dictionary alows you to associate more than one value with each key.

Storing lists in a dictionary

```
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

Nesting — A dictionary of dictionaries

You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.

Storing dictionaries in a dictionary

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },
    }

for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']

    print("\tFull name: " + full_name.title())
    print("\tLocation: " + location.title())
```

Levels of nesting

Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.

Using an OrderedDict

Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an OrderedDict.

Preserving the order of keys and values

```
from collections import OrderedDict

# Store each person's languages, keeping
#  track of who respoded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
#  were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

Generating a million dictionaries

You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.

A million aliens

```
aliens = []

# Make a million green aliens, worth 5 points
#  each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that idenitfy the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.

Checking for equality

A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions

You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Boolean values

A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True
can_edit = False
```

If statements

Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 5
else:
    price = 10

print("Your cost is $" + str(price) + ".")
```

Conditional tests with lists

You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

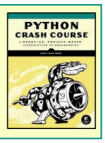
Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```


Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Conditional tests with lists (cont.)

Testing if a value is not in a list

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'

if user not in banned_users:
    print("You can play!")
```

Checking if a list is empty

```
players = []

if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

Accepting input

You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.

Simple input

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

Accepting numerical input

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

Accepting input in Python 2.7

Use raw_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While loops

A while loop repeats a block of code as long as a condition is True.

Counting to 5

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

While loops (cont.)

Letting the user choose when to quit

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Using a flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to exit a loop

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

Accepting input with Sublime Text

Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.

Breaking out of loops

You can use the break statement and the continue statement with any of Python's loops. For example you can use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.

While loops (cont.)

Using continue in a loop

```
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

Avoiding infinite loops

Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.

An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

Removing all instances of a value from a list

The remove() method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.

Removing all cats from a list of pets

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Functions

What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

Defining a function

The first line of a function is its definition, marked by the keyword def. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.

To call a function, give the name of the function followed by a set of parentheses.

Making a function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing information to a function

Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.

Passing a single argument

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Using keyword arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

Default values

You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.

Using a default value

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

Using None to make an argument optional

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

Return values

A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.

Returning a single value

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

Returning a dictionary

```
def build_person(first, last):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

Returning a dictionary with optional values

```
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

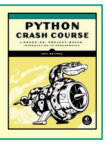
Visualizing functions

Try running some of these examples on pythontutor.com.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Passing a list to a function

You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.

Passing a list as an argument

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Allowing a function to modify a list

The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

Preventing a function from modifying a list

The following example is the same as the previous one, except the original list is unchanged after calling print_models().

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

Passing an arbitrary number of arguments

*Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the * operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition.*

*The ** operator allows a parameter to collect an arbitrary number of keyword arguments.*

Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
           'onions', 'extra cheese')
```

Collecting an arbitrary number of keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
#   of information.
user_0 = build_profile('albert', 'einstein',
         location='princeton')
user_1 = build_profile('marie', 'curie',
         location='paris', field='chemistry')

print(user_0)
print(user_1)
```

What's the best way to structure a function?

As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.

Modules

You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)

Storing a function in a module

File: pizza.py

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

Importing an entire module

File: making_pizzas.py
Every function in the module is available in the program file.

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

Importing a specific function

Only the imported functions are available in the program file.

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

Giving a module an alias

```
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

Giving a function an alias

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

Importing all functions from a module

Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.

```
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.

Creating and using a class

Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is stored in variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an object from a class

```
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

Modifying attributes

You can modify an attribute's value directly, or you can write methods that manage updating values more carefully.

Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
            <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

In Python class names are written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should still be named in lowercase with underscores.

Class inheritance

If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.

To inherit from another class include the name of the parent class in parentheses when defining the new class.

The __init__() method for a child class

```
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 70
        # Charge level in %.
        self.charge_level = 0
```

Adding new methods to the child class

```
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```
my_ecar = ElectricCar('tesla', 'model s', 2016)

my_ecar.charge()
my_ecar.drive()
```

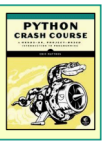
Finding your workflow

There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it – if your first attempt doesn't work, try a different approach.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse



Class inheritance (cont.)

Overriding parent methods

```
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

A class can have objects as attributes. This allows classes to work together to model complex situations.

A Battery class

```
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.

Storing classes in a file

car.py

```
"""Represent gas and electric cars."""

class Car():
    """A simple attempt to model a car."""
    --snip--

class Battery():
    """A battery for an electric car."""
    --snip--

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module

my_cars.py

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s',
2016)
my_tesla.charge()
my_tesla.drive()
```

Importing an entire module

```
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
        'tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing all classes from a module

(Don't do this, but recognize it when you see it.)

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Classes in Python 2.7

Classes should inherit from object

```
class ClassName(object):
```

The Car class in Python 2.7

```
class Car(object):
```

Child class __init__() method is different

```
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

The ElectricCar class in Python 2.7

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
                make, model, year)
```

Storing objects in a list

A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.

Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.

A fleet of rental cars

```
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The with statement makes sure the file is closed properly when the program has finished accessing the file.

Reading an entire file at once

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line

Each line that's read from the file has a newline character at the end of the line, and the print function adds its own newline character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file

Passing the 'w' argument to open() tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.

Writing to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths

When Python runs the open() function, it looks for the file in the same directory where the program that's being excuted is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.

Opening a file from a subfolder

```
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows

Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.

```
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block

When you think an error may occur, you can write a try-except block to handle the exception that might be raised. The try block tells Python to try running some code, and the except block tells Python what to do if the code results in a particular kind of error.

Handling the ZeroDivisionError exception

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

Knowing which exception to handle

It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.

Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse

The else block

The try block should only contain code that may cause an error. Any code that depends on the try block running successfully should be placed in the else block.

Using an else block

```
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes from user input

Without the except block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.

```
"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break
    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.

It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.

Failing silently

Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the pass statement in an else block allows you to do this.

Using the pass statement in an else block

```
f_names = ['alice.txt', 'siddhartha.txt',
        'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    try:
        with open(f_name) as f_obj:
            lines = f_obj.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = "{0} has {1} lines.".format(
            f_name, num_lines)
        print(msg)
```

Avoid bare except blocks

Exception-handling code should catch specific exceptions that you expect to happen during your program's execution. A bare except block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.

If you want to use a try block and you're not sure which exception to catch, use Exception. It will catch most exceptions, but still allow you to interrupt programs intentionally.

Don't use bare except blocks

```
try:
    # Do something
except:
    pass
```

Use Exception instead

```
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

The json module allows you to dump simple Python data structures into a file, and load the data from that file the next time the program runs. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.

Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.

Using json.dump() to store data

```
"""Store some numbers."""

import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

Using json.load() to read data

```
"""Load some previously stored numbers."""

import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

Making sure the stored data exists

```
import json

f_name = 'numbers.json'

try:
    with open(f_name) as f_obj:
        numbers = json.load(f_obj)
except FileNotFoundError:
    msg = "Can't find {0}.".format(f_name)
    print(msg)
else:
    print(numbers)
```

Practice with exceptions

Take a program you've already written that prompts for user input, and add some error-handling code to the program.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Testing Your Code

Why test your code?

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in the situations it's designed to handle, and also when people use your programs in unexpected ways. Writing tests gives you confidence that your code will work correctly as more people begin to use your programs. You can also add new features to your programs and know that you haven't broken existing behavior.

A unit test verifies that one specific aspect of your code works as it's supposed to. A test case is a collection of unit tests which verify your code's behavior in a wide variety of situations.

Testing a function: A passing test

Python's unittest module provides tools for testing your code. To try it out, we'll create a function that returns a full name. We'll use the function in a regular program, and then build a test case for the function.

A function to test

Save this as full_names.py

```
def get_full_name(first, last):
    """Return a full name."""
    full_name = "{0} {1}".format(first, last)
    return full_name.title()
```

Using the function

Save this as names.py

```
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

Testing a function (cont.)

Building a testcase with one unit test

To build a test case, make a class that inherits from `unittest.TestCase` *and write methods that begin with* `test_`. *Save this as test_full_names.py*

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

unittest.main()
```

Running the test

Python reports on each unit test in the test case. The dot reports a single passing test. Python informs us that it ran 1 test in less than 0.001 seconds, and the OK lets us know that all unit tests in the test case passed.

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

Testing a function: A failing test

Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you need to modify the code so the existing behavior still works.

Modifying the function

We'll modify `get_full_name()` *so it handles middle names, but we'll do it in a way that breaks existing behavior.*

```
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = "{0} {1} {2}".format(first,
                                     middle, last)
    return full_name.title()
```

Using the function

```
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

A failing test (cont.)

Running the test

When you change your code, it's important to run your existing tests. This will tell you whether the changes you made affected existing behavior.

```
E
======================================================
ERROR: test_first_last (__main__.NamesTestCase)
Test names like Janis Joplin.
------------------------------------------------------
Traceback (most recent call last):
  File "test_full_names.py", line 10,
    in test_first_last
    'joplin')
TypeError: get_full_name() missing 1 required
  positional argument: 'last'

------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Fixing the code

When a test fails, the code needs to be modified until the test passes again. (Don't make the mistake of rewriting your tests to fit your new code.) Here we can make the middle name optional.

```
def get_full_name(first, last, middle=''):
    """Return a full name."""
    if middle:
        full_name = "{0} {1} {2}".format(first,
                                         middle, last)
    else:
        full_name = "{0} {1}".format(first,
                                     last)
    return full_name.title()
```

Running the test

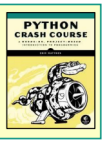
Now the test should pass again, which means our original functionality is still intact.

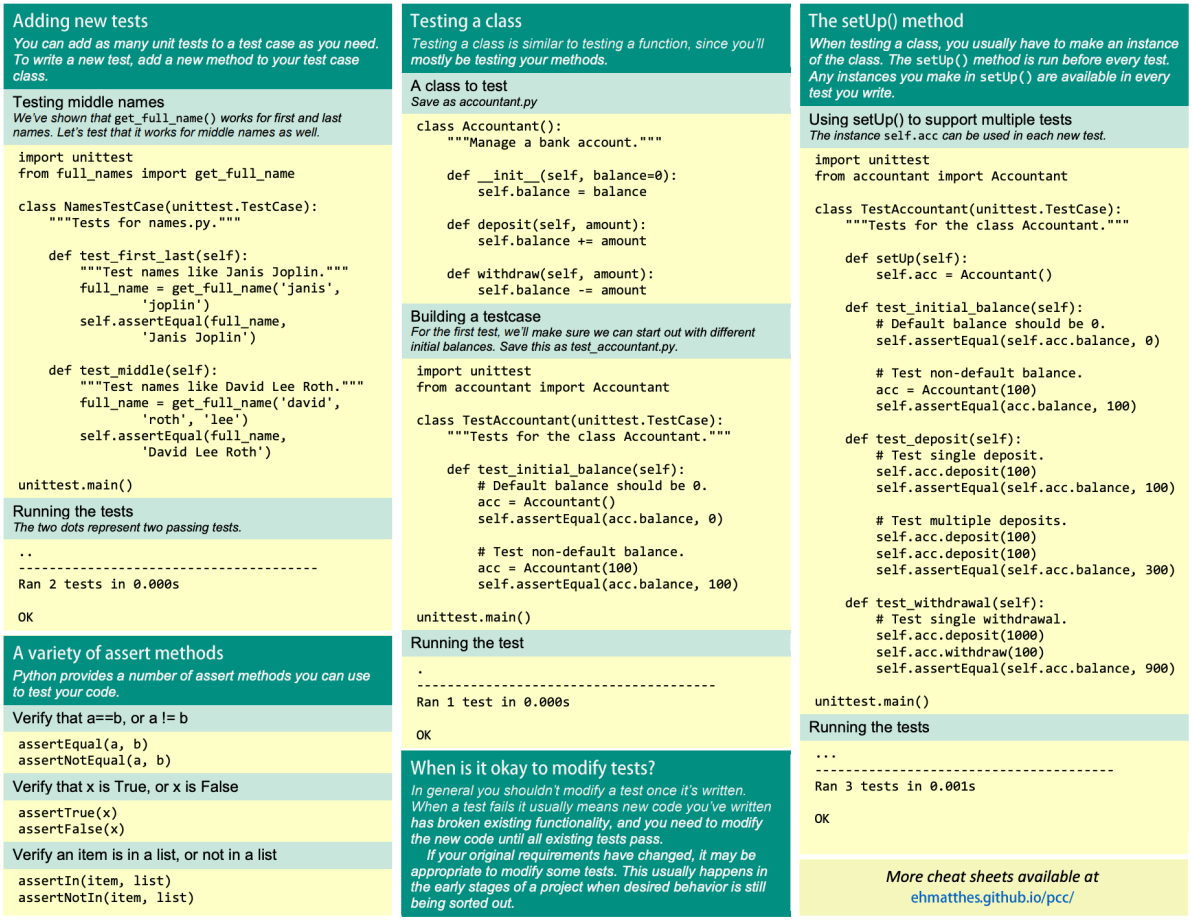
```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```


Python Crash Course

Covers Python 3 and Python 2

nostarchpress.com/pythoncrashcourse





Adding new tests

You can add as many unit tests to a test case as you need. To write a new test, add a new method to your test case class.

Testing middle names

We've shown that `get_full_name()` works for first and last names. Let's test that it works for middle names as well.

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

    def test_middle(self):
        """Test names like David Lee Roth."""
        full_name = get_full_name('david',
                'roth', 'lee')
        self.assertEqual(full_name,
                'David Lee Roth')

unittest.main()
```

Running the tests

The two dots represent two passing tests.

```
..
----------------------------------------
Ran 2 tests in 0.000s

OK
```

A variety of assert methods

Python provides a number of assert methods you can use to test your code.

Verify that a==b, or a != b

```
assertEqual(a, b)
assertNotEqual(a, b)
```

Verify that x is True, or x is False

```
assertTrue(x)
assertFalse(x)
```

Verify an item is in a list, or not in a list

```
assertIn(item, list)
assertNotIn(item, list)
```

Testing a class

Testing a class is similar to testing a function, since you'll mostly be testing your methods.

A class to test

Save as accountant.py

```
class Accountant():
    """Manage a bank account."""

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

Building a testcase

For the first test, we'll make sure we can start out with different initial balances. Save this as test_accountant.py.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def test_initial_balance(self):
        # Default balance should be 0.
        acc = Accountant()
        self.assertEqual(acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

unittest.main()
```

Running the test

```
.
----------------------------------------
Ran 1 test in 0.000s

OK
```

When is it okay to modify tests?

In general you shouldn't modify a test once it's written. When a test fails it usually means new code you've written has broken existing functionality, and you need to modify the new code until all existing tests pass.

If your original requirements have changed, it may be appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out.

The setUp() method

When testing a class, you usually have to make an instance of the class. The `setUp()` method is run before every test. Any instances you make in `setUp()` are available in every test you write.

Using setUp() to support multiple tests

The instance `self.acc` can be used in each new test.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def setUp(self):
        self.acc = Accountant()

    def test_initial_balance(self):
        # Default balance should be 0.
        self.assertEqual(self.acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

    def test_deposit(self):
        # Test single deposit.
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 100)

        # Test multiple deposits.
        self.acc.deposit(100)
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 300)

    def test_withdrawal(self):
        # Test single withdrawal.
        self.acc.deposit(1000)
        self.acc.withdraw(100)
        self.assertEqual(self.acc.balance, 900)

unittest.main()
```

Running the tests

```
...
----------------------------------------
Ran 3 tests in 0.001s

OK
```

More cheat sheets available at
ehmatthes.github.io/pcc/

We will use Codecademy (https://www.codecademy.com/learn/learn-python-3) to learn Python throughout the course.

5 Version Control

Version control systems are essential tools in software development, designed to manage changes to computer programs, documents, large websites, or other collections of information.

5.1 What is Version Control?

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It allows multiple users to edit and manage the same files simultaneously, providing mechanisms to navigate and resolve conflicts when they occur.

5.2 Git: The Leading Version Control System

GitHub is a web-based platform that uses Git for version control. It not only hosts your source code but also offers a rich feature set for project management, including issue tracking, feature requests, task management, and wikis for documentation. GitHub facilitates transparency and collaboration in software development projects by providing tools that enable team members to discuss changes, review code, and contribute from anywhere.

5.3 GitHub: A Hub for Collaborative Projects

GitHub is a web-based platform that uses Git for version control. It facilitates more than just code hosting; it provides a collaborative environment where developers can review code, manage projects, and build software alongside millions of other developers.

01 Git configuration

`git config --global user.name "Your Name"`	Set the name that will be attached to your commits and tags.
`git config --global user.email "you@example.com"`	Set the e-mail address that will be attached to your commits and tags.
`git config --global color.ui auto`	Enable some colorization of Git output.

02 Starting a project

`git init [project name]`	Create a new local repository in the current directory. If **[project name]** is provided, Git will create a new directory named **[project name]** and will initialize a repository inside it.
`git clone <project url>`	Downloads a project with the entire history from the remote repository.

03 Day-to-day work

`git status`	Displays the status of your working directory. Options include new, staged, and modified files. It will retrieve branch name, current commit identifier, and changes pending commit.
`git add [file]`	Add a file to the **staging** area. Use. in place of the full file path to add all changed files from the **current directory** down into the **directory tree.**
`git diff [file]`	Show changes between **working directory** and **staging area.**
`git diff --staged [file]`	Shows any changes between the **staging area** and the **repository.**
`git checkout -- [file]`	Discard changes in **working directory.** This operation is **unrecoverable**.
`git reset [<path>...]`	Revert some paths in the index (or the whole index) to their state in **HEAD.**
`git commit`	Create a new commit from changes added to the **staging area.** The **commit** must have a message!

06 Inspect history

`git log [-n count]`	List commit history of current branch. **-n count** limits list to last **n** commits.
`git log --oneline --graph --decorate`	An overview with reference labels and history graph. One commit per line.
`git log ref..`	List commits that are present on the current branch and not merged into **ref**. A **ref** can be a branch name or a tag name.
`git log ..ref`	List commit that are present on **ref** and not merged into current branch.
`git reflog`	List operations (e.g. checkouts or commits) made on local repository.

07 Tagging commits

`git tag`	List all tags.
`git tag [name] [commit sha]`	Create a tag reference named **name** for current commit. Add **commit sha** to tag a specific commit instead of current one.
`git tag -a [name] [commit sha]`	Create a tag object named **name** for current commit.
`git tag -d [name]`	Remove a tag from local repository.

08 Reverting changes

`git reset [--hard] [target reference]`	Switches the current branch to the **target reference**, leaving a difference as an uncommitted change. When **--hard** is used, all changes are discarded. It's easy to lose uncommitted changes with **--hard.**
`git revert [commit sha]`	Create a new commit, reverting changes from the specified commit. It generates an **inversion** of changes.

`git rm [file]`	Remove file from **working directory** and **staging area.**

04 Storing your work

`git stash`	Put current changes in your **working directory** into **stash** for later use.
`git stash pop`	Apply stored **stash** content into **working directory,** and clear **stash.**
`git stash drop`	Delete a specific **stash** from all your previous **stashes.**

05 Git branching model

`git branch [-a]`	List all local branches in repository. With **-a**: show all branches (with remote).
`git branch [branch_name]`	Create new branch, referencing the current **HEAD**.
`git rebase [branch_name]`	Apply commits of the current working branch and apply them to the HEAD of [branch] to make the history of your branch more linear.
`git checkout [-b] [branch_name]`	Switch working directory to the specified branch. With **-b**: Git will create the specified branch if it does not exist.
`git merge [branch_name]`	Join specified **[branch_name]** branch into your current branch (the one you are on currently).
`git branch -d [branch_name]`	Remove selected branch, if it is already merged into any other. **-D** instead of **-d** forces deletion.

Commit	a state of the code base
Branch	a reference to a commit; can have a **tracked upstream**
Tag	a reference (standard) or an object (annotated)
HEAD	a place where your **working directory** is now

09 Synchronizing repositories

`git fetch [remote]`	Fetch changes from the **remote**, but not update tracking branches.
`git fetch --prune [remote]`	Delete remote Refs that were removed from the **remote** repository.
`git pull [remote]`	Fetch changes from the **remote** and merge current branch with its upstream.
`git push [--tags] [remote]`	Push local changes to the **remote**. Use **--tags** to push tags.
`git push -u [remote] [branch]`	Push local branch to **remote** repository. Set its copy as an upstream.

10 Git installation

For GNU/Linux distributions, Git should be available in the standard system repository. For example, in Debian/Ubuntu please type inthe terminal:

```
sudo apt-get install git
```

If you need to install Git from source, you can get it from **git-scm.com/downloads**.

An excellent Git course can be found in the great Pro Git book by Scott Chacon and Ben Straub. The book is available online for free at **git-scm.com/book**.

11 Ignoring files

```
cat  <<EOF >  .gitignore
/logs/*
!logs/.gitkeep
/tmp
*.swp
EOF
```

To ignore files, create a .gitignore file in your repository with a line for each pattern. File ignoring will work for the current and sub directories where .gitignore file is placed. In this example, all files are ignored in the logs directory (excluding the .gitkeep file), whole tmp directory and all files *.swp.

Figure 4

[4] https://about.gitlab.com/images/press/git-cheat-sheet.pdf

Core Features of GitHub:

- **Repositories**: These are used to host project files and keep track of the version history.
- **Forks**: Copy another user's repository to your account to modify it without affecting the original.
- **Pull Requests**: Suggest changes to the repository owner, which can then be merged into the main branch. Let you tell others about changes you've pushed to a GitHub repository. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before the changes are merged into the base branch.
- **Issues**: Tracking tasks, enhancements, and bugs for projects.

6 Architecture of Computers

Computer architecture encompasses a wide range of topics foundational to understanding how computers function at a hardware and software level. This report will explore various critical components, including Boolean logic, truth tables, number systems, memory, operating systems, CPUs, GPUs, assembly code, and security considerations like buffer overflow.

6.1 Definitions

Here are some brief definitions:

- **Boolean Logic:** Boolean logic forms the basis of modern computing. It involves true/false values (binary) and is used in digital circuits to carry out operations. Truth tables visualise and predict the outcome of these logical operations and are essential for designing and understanding digital circuits.
- **Binary Number:** Computers operate using the binary number system, which consists of two binary digits, 0 and 1. Programmers also use other number systems, such as hexadecimal and octal, for various computing applications.
- **Memory:** Memory is where data is stored temporarily or permanently in a computer. Key types include RAM (Random Access Memory), ROM (Read-Only Memory), and cache, each serving different functions and performance roles.
- **OS:** The operating system (OS) manages a computer's hardware resources and provides services for computer programs. It acts as an intermediary between applications and the computer hardware.
- **CPU:** The CPU (Central Processing Unit) is the primary component of a computer that performs most of the processing inside. CPUs handle basic instructions from a computer's hardware and software.
- **GPU:** GPUs (Graphics Processing Units) are specialised electronic circuits designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display.
- **Assembly Language:** Assembly language is a low-level programming language with strong correspondence between the language and the architecture's machine code instructions. It is used to write programs that interact directly with the hardware.
- **Buffer Overflow:** A buffer overflow occurs when a program writes more data to a buffer than it is designed to hold. This can lead to unpredictable behaviour, including the execution of malicious code.

6.2 Boolean Logic

Boolean logic is a form of algebra where all values are either True or False. It is fundamental to computer science because it underpins the operations of digital circuits and programming conditions.

Key Boolean Operations:

- **AND**: True if both operands are true.
- **OR**: True if at least one operand is true.
- **NOT**: True if the operand is false; false if it is true.

6.2.1 Truth Tables

Truth tables are used to visualise and define the output of Boolean operations for all possible input combinations. For example, a truth table for the AND operation would look like this:

A	B	A AND B
F	F	F
F	T	F
T	F	F
T	T	T

Truth tables are essential for designing digital circuits and debugging logical expressions in software.

6.2.2 Logic Gates

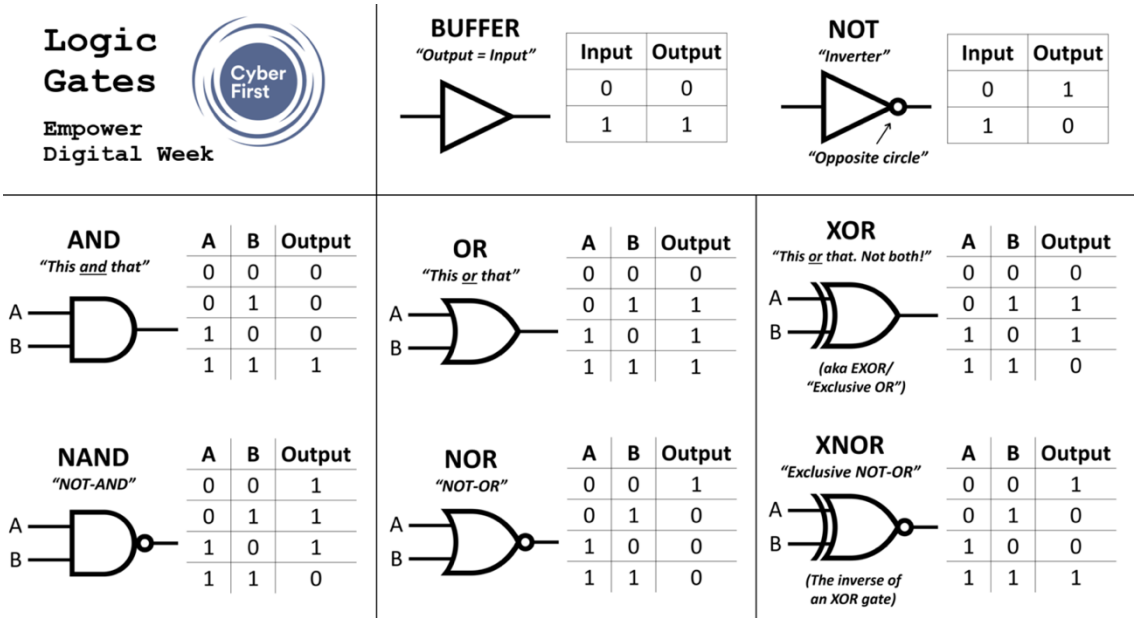


Figure 5

[5] https://www.ncsc.gov.uk/static-assets/documents/ECW21-NCSC-Logic-Gate-Cheat-Sheet.pdf

6.3 Number Systems

Computers use various number systems, primarily binary, but also hexadecimal and decimal, for processing and memory storage.

- **Binary (Base-2)**: Used directly by computers, representing values using only 0 and 1.
- **Decimal (Base-10)**: The standard system for human-centric numbering, which uses digits from 0 to 9.
- **Hexadecimal (Base-16)** is a human-friendly representation of binary data used in computing. Digits range from 0 to 9 and A to F.

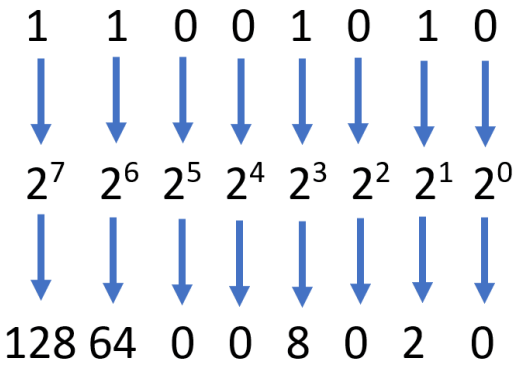


Figure 6

6.4 Memory

Memory is where data is stored temporarily or permanently in a computer. The primary types are **RAM (Random Access Memory)**, which is fast, volatile memory used to store data that a CPU needs while performing tasks, and **Hard Drives and SSDs (Solid-State Drives)**, which are used for long-term data storage. SSDs are faster than hard drives because they use flash memory and have no moving parts.

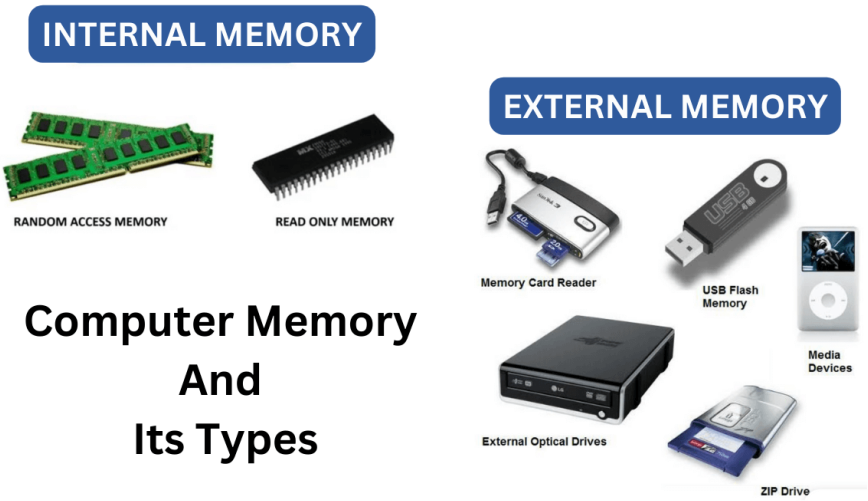
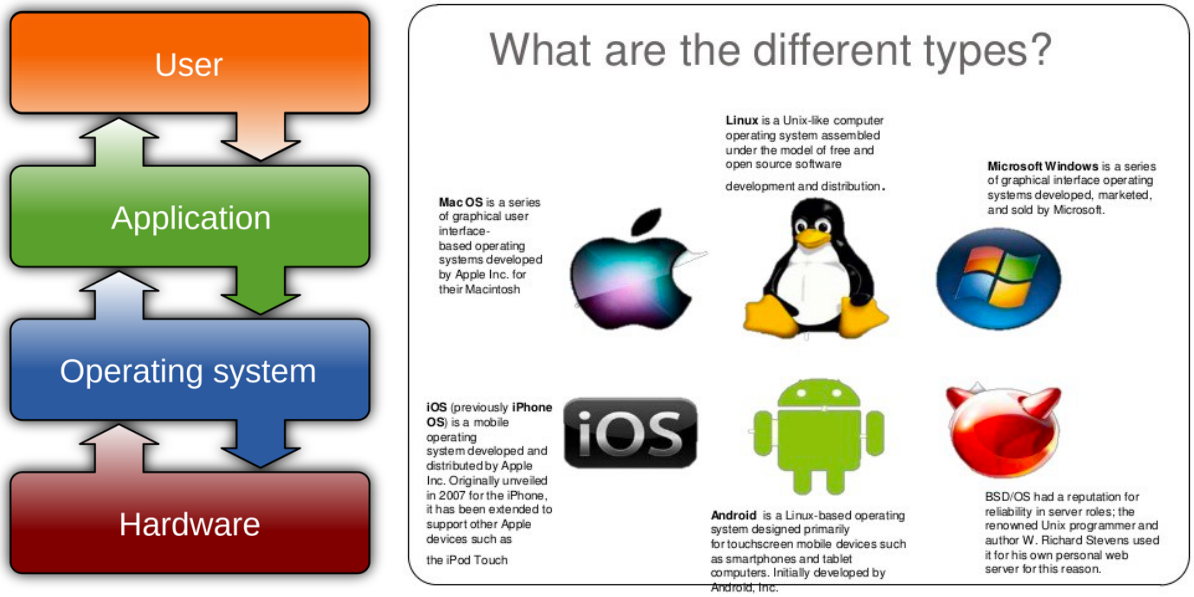


Figure 7

[6] https://www.lifewire.com/how-to-read-binary-4692830
[7] https://www.careerpower.in/school/computer/computer-memory-and-its-types

6.5 Operating Systems

Operating systems (OS) manage a computer's hardware resources and provide services for computer programs. An OS performs basic tasks such as recognising input from the keyboard, sending output to the display screen, keeping track of files and directories, and controlling peripheral devices. Examples include Microsoft Windows, macOS, Linux, and UNIX.



Figures 8, 9

6.6 CPUs and GPUs

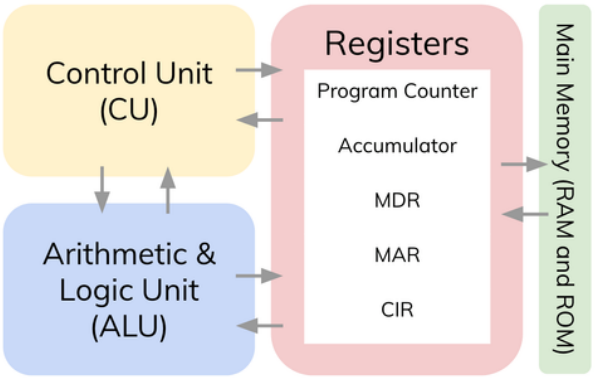


Figure 10

- **CPU (Central Processing Unit):** Known as the computer's brain, the CPU performs most of the processing inside a computer. It handles basic instructions and allocates more complex tasks to other specific chips or peripherals.
- **GPU (Graphics Processing Unit):** GPUs are designed to render graphics and images by processing vast chunks of data in parallel. While initially intended for graphical tasks,

[8] https://en.wikipedia.org/wiki/File_system
[9] https://www.goconqr.com/mindmap/11920038/operating-systems
[10] https://senecalearning.com/en-GB/revision-notes/a-level/computer-science/ocr/1-1-4-cpu-architecture

GPUs are increasingly used for machine learning and data analysis due to their efficiency in parallel processing.

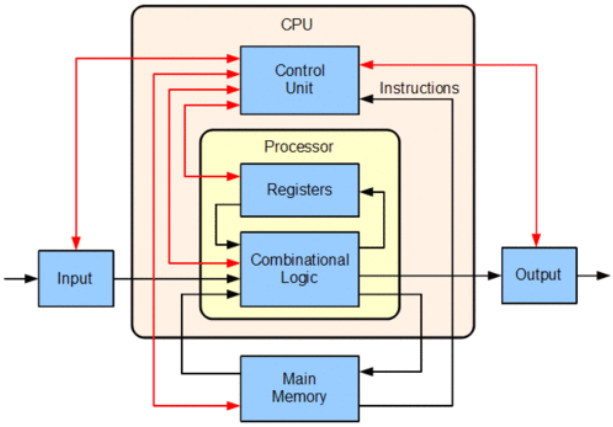


Figure 11

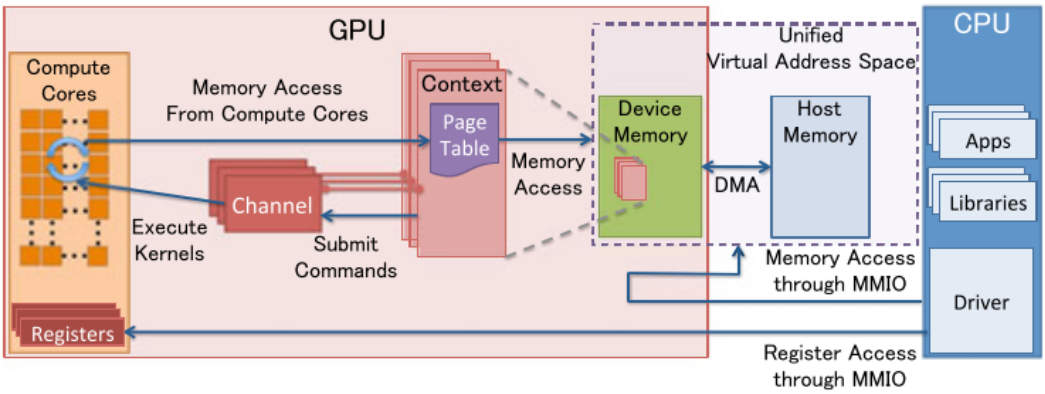


Figure 12

6.7 Assembly Code

Assembly code is a low-level programming language that's closely correlated with a computer's machine language instructions. It is specific to a particular computer architecture. Assembly is used where high performance is essential or in systems that require direct hardware manipulation.

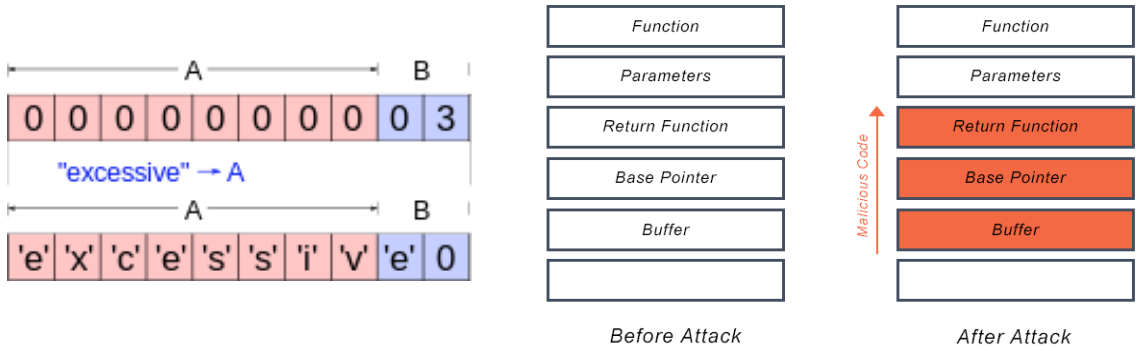
Example of Assembly Code (x86 architecture):
mov eax, 1 ; *Move 1 into the EAX register*
add eax, 2 ; *Add 2 to the EAX register*

6.8 Buffer Overflow

A buffer overflow occurs when data exceeds a buffer's storage capacity, overwriting adjacent memory locations. This can cause erratic program behaviour, including access violations,

[11] https://tok.fandom.com/wiki/Computer_science
[12] https://insujang.github.io/2017-04-27/gpu-architecture-overview/

incorrect results, or a crash. More critically, buffer overflow vulnerabilities can be exploited to execute arbitrary code, potentially allowing a hacker to take control of the system.



Figures 13, 14

7 Database Systems

This comprehensive overview of database systems explores the architecture, design, and operation of databases, with a focus on relational databases. We also delve into big data, database management systems, data privacy, visualisation techniques, and the ethical considerations in database usage, including security vulnerabilities like SQL injections.

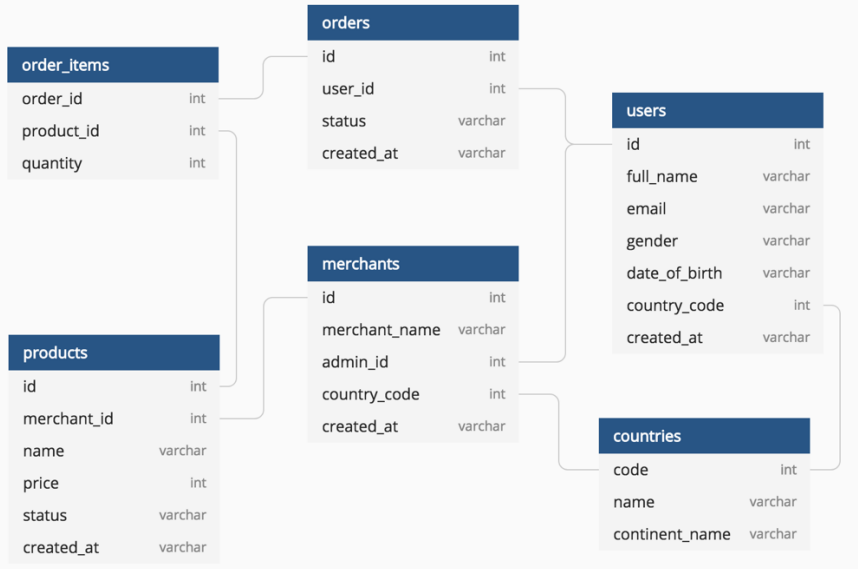


Figure 15

7.1 Definitions

Here are some brief definitions:

- **Database:** A database system is an organised collection of data, generally stored and accessed electronically from a computer system. The architecture of a database system is crucial as it defines the structure and methods used to store, retrieve, and manage data.

[13] https://en.wikipedia.org/wiki/Buffer_overflow
[14] https://avinetworks.com/glossary/buffer-overflow/
[15] https://www.holistics.io/blog/top-5-free-database-diagram-design-tools/

- **Relational Algebra:** Relational algebra is a set of operations used to manipulate relations and provides a theoretical foundation for relational databases and SQL.
- **ER Modelling:** ER modelling is a data modelling technique for visually representing the relationships between different data models in a relational database.
- **Normalisation:** Normalisation involves organising data in a database to reduce redundancy and improve data integrity. It ensures that databases are structured efficiently and logically.
- **Big Data:** Big data involves processing high volumes of low-density, unstructured data. This data can be of unknown value, such as Twitter data feeds, clickstreams on a webpage or a mobile app, or sensor-enabled equipment.
- **DBMS:** A Database Management System (DBMS) is crucial in big data for storing, processing, and analysing large datasets efficiently.
- **Data Privacy:** Data privacy concerns the proper handling, processing, storage, and usage of personal information in compliance with regulations and laws. These include GDPR (General Data Protection Regulation) in the European Union and CCPA (California Consumer Privacy Act) in California, USA.

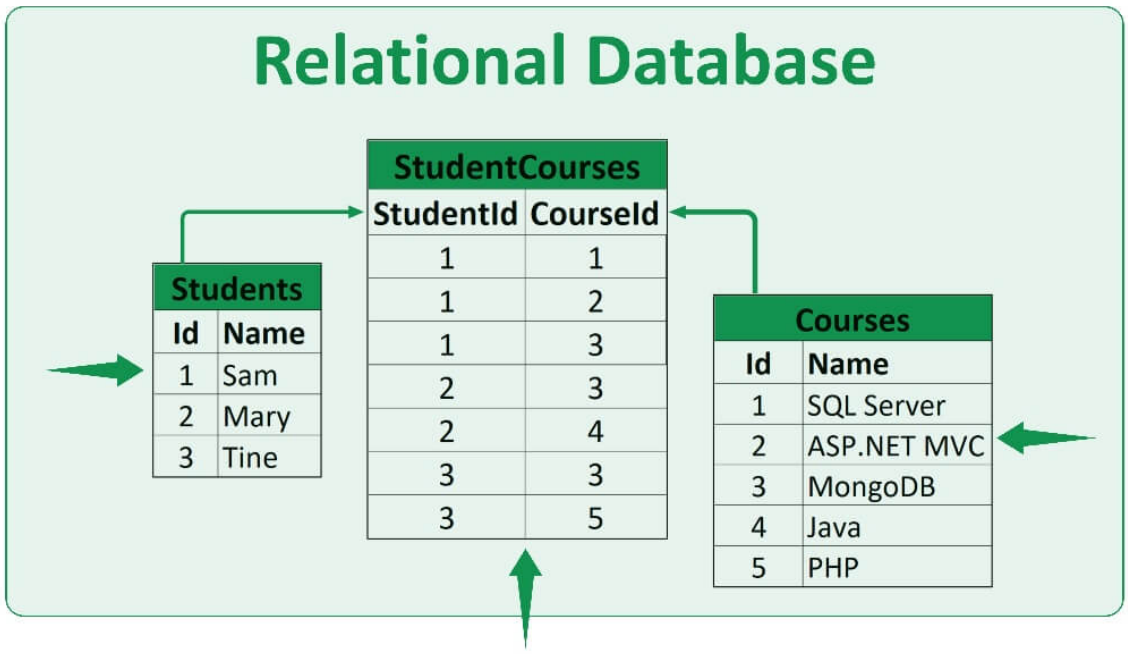


Figure 16

7.2 Boolean Logic

Boolean Logic is fundamental to the search functions of online search engines and library databases.

- **AND:** The results will contain both words. AND is also implied by a blank space between words. For example, Results from Geriatric AND Driving will contain both the words Geriatric and Driving.

[16] https://www.pragimtech.com/blog/mongodb-tutorial/relational-and-non-relational-databases/

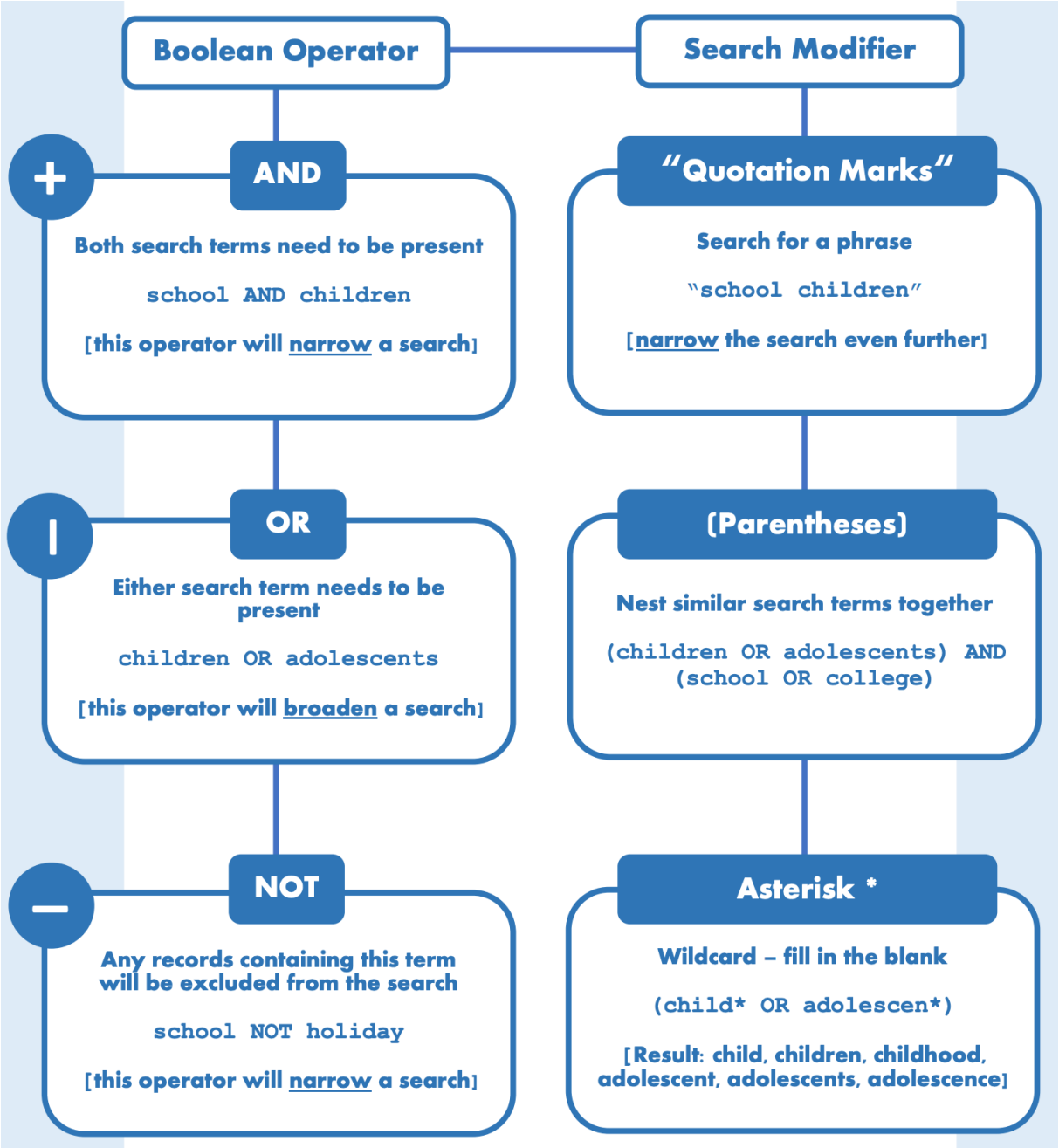


Figure 17

- **OR:** The results will contain one or more of the words. E.g. Results from Geriatric OR Elderly will contain Geriatric or Elderly or both.
- **NOT:** Results will not include the search term. E.g. Results from Elderly NOT "Middle Aged" will contain Elderly but not "Middle Aged".
- **Search Modifiers:** Quotation Marks " ". Results will contain the words together as specified. E.g. Results from "Middle Aged" will contain Middle and Aged as a phrase.
- **Asterisk *:** Results will contain words that begin with the search term. For example, results from Driv* will contain Drive, Driver, Driving, Driven.

[17] https://arc-w.nihr.ac.uk/Wordpress/wp-content/uploads/2021/09/Boolean-Cheat-Sheet.pdf

- **Parentheses ( ):** Search terms can be grouped using parentheses. For example, results from (Geriatric OR Senior) AND "Driving Cessation" will contain "Driving Cessation" and either or both the words Geriatric or Senior.
- **Search Strings:** Boolean Operators and Modifiers can be combined to form search strings. E.g. A search string about driving cessation amongst seniors might look like this: "Driving Cessation" AND (geriatric OR senior OR "older adults")

Quick Tips
- Use AND or NOT to narrow results; use OR to broaden results
- Some applications do not support the Asterisk Modifier. Instead, construct OR statements to search all variations
- Record each search string to avoid duplication

7.3 Database Architecture

Database architecture refers to the structured design of databases, including the methods and models that dictate how data is stored, accessed, and managed. The architecture can vary significantly depending on the type of database and the specific requirements of the system.

7.3.1 Relational Algebra

SQL is the standard language for managing relational databases. It uses relational algebra as its underlying framework to manipulate and retrieve data. Understanding SQL is essential for interacting with databases, from basic data entry to complex queries and administrative tasks.

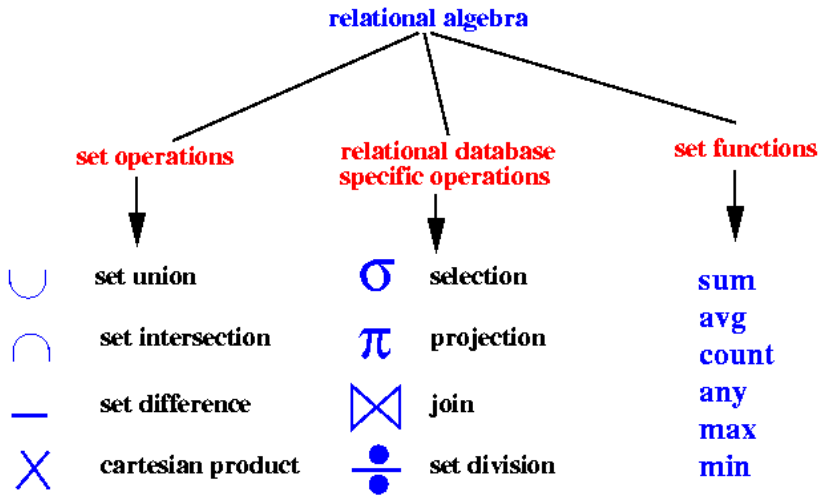


Figure 18

Relational algebra is a set of relational database operations that enable constructing queries. The fundamental operations are:

- **Select**: Extracts rows from a table that meet a specific criterion.
- **Project**: Selects columns from a table and discards the others.
- **Join**: Combines rows from two or more tables based on a related column.

[18] https://medium.com/@sahirnambiar/relational-algebra-the-underpinnings-of-sql-74959481231a

- **Union, Intersection, Difference**: These operations are used to combine or differentiate sets of tuples.

These operations form the basis for structured query language (SQL), which is used to interact with relational databases.

7.3.2 ER Modelling

Entity-relationship (ER) modelling is a graphical approach to database design. It involves defining entities (things about which data is stored) and relationships (associations between entities). An ER diagram helps visualise the database structure, showing entity attributes and the links between different entities, facilitating a clearer understanding of the database's layout.

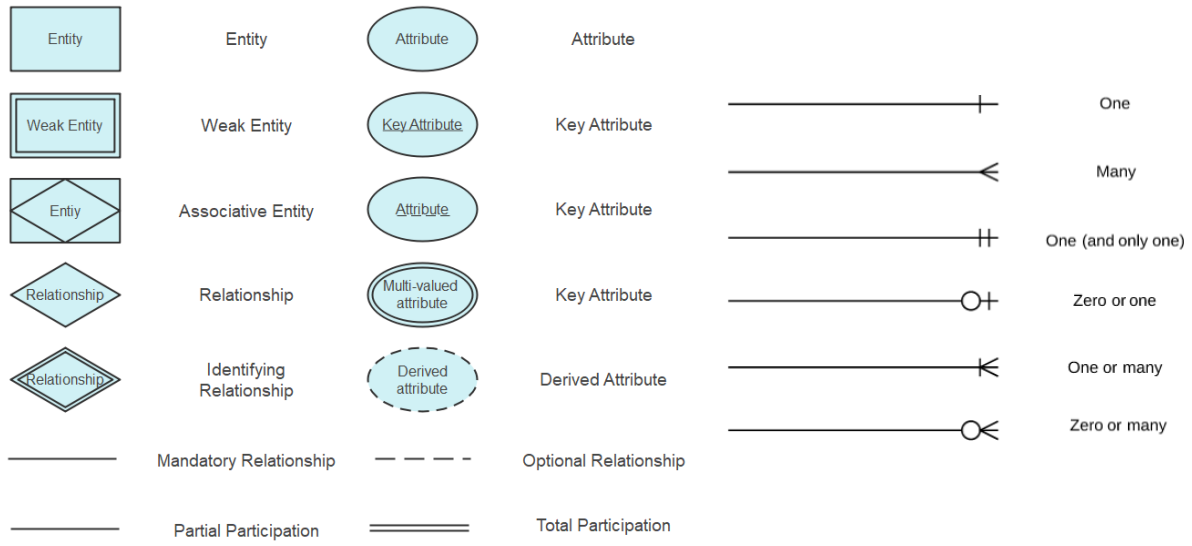


Figure 19, 20

7.3.3 Normalisation and Database Design

Normalisation is the process of organising data in a database to reduce redundancy and improve data integrity. The stages of normalization, progressively reduce data redundancy and improve data integrity by removing partial, transitive, and other forms of dependencies.

7.3.4 Big Data and Database Management Systems

Big data refers to extremely large data sets that may be analysed computationally to reveal patterns, trends, and associations, especially relating to human behaviour and interactions. Database management systems (DBMS) like MySQL, PostgreSQL, and MongoDB support big data by providing robust, scalable solutions for data storage, manipulation, and retrieval.

[19] https://www.edrawsoft.com/er-diagram-symbols.html
[20] https://www.lucidchart.com/pages/ER-diagram-symbols-and-meaning

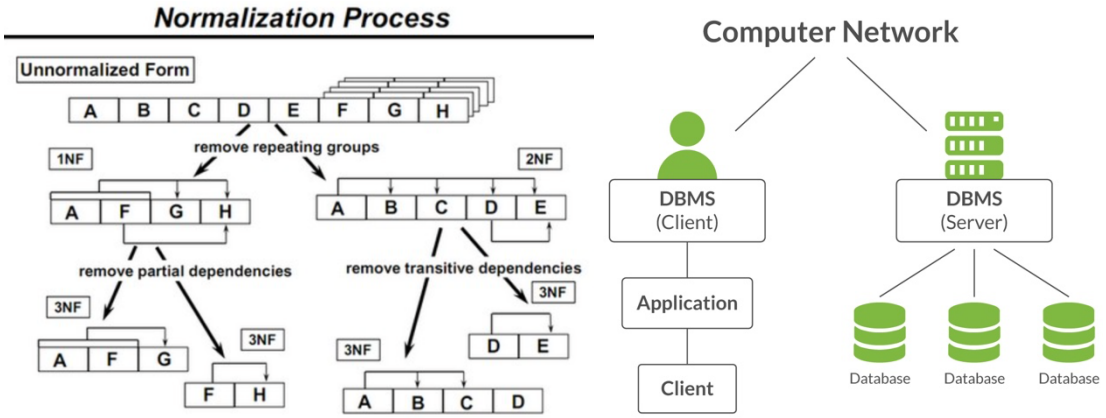


Figure 21, 22

7.3.5 Maintaining Data Privacy Regulations

Maintaining data privacy is crucial in database management. Regulations such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA) mandate strict guidelines on data collection, storage, and sharing. Databases must be designed to comply with these regulations to protect user data from unauthorized access and breaches.

7.3.6 Visualising Databases

Database visualization involves using tools and software to visually navigate and display the structure of databases and their interconnections. Tools like Microsoft SQL Server Management Studio, Oracle SQL Developer, and DBeaver provide graphical interfaces to enhance the visibility of complex data structures, making management more intuitive and accessible.

7.4 SQL Injections and Vulnerabilities

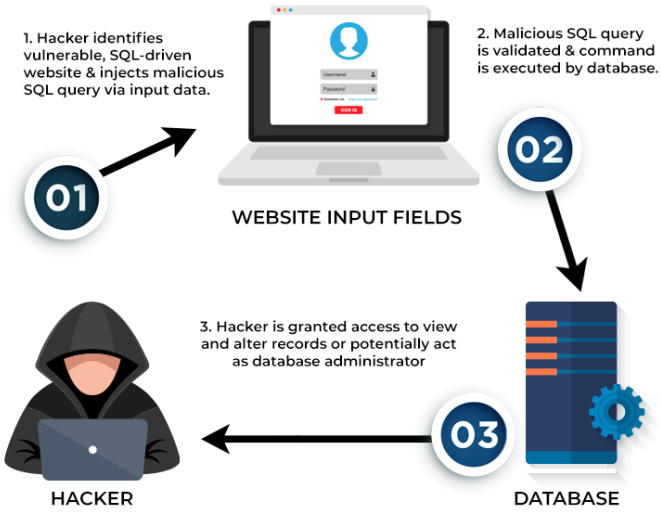


Figure 23

[21] https://learn.saylor.org/mod/page/view.php?id=23139
[22] https://kb.n0c.com/en/glossaire/dbms/
[23] https://www.spiceworks.com/it-security/application-security/articles/what-is-sql-injection/

SQL injection is a common attack technique that exploits vulnerabilities in an application's software by injecting malicious SQL statements into a query. For example:

SELECT * FROM users WHERE username = 'admin'--' AND password = 'password';

Here, -- comments out the rest of the SQL statement, potentially allowing unauthorized access. To mitigate this, developers use prepared statements and parameterized queries, which ensure that SQL commands are safely separated from data inputs.

8 Cryptographic Techniques

Cryptography is the science of protecting information by transforming it into a secure format. This process, known as encryption, makes the information unreadable to anyone except those possessing special knowledge, usually referred to as a key.

We will explore cryptographic techniques that are pivotal for ensuring the confidentiality, integrity, and authenticity of digital information. We will delve into the mechanics of encryption and decryption, differentiate between symmetric and asymmetric encryption, and illustrate how messages are encrypted and decrypted both mathematically and through Python coding examples.

8.2 Overview of Encryption and Decryption Process

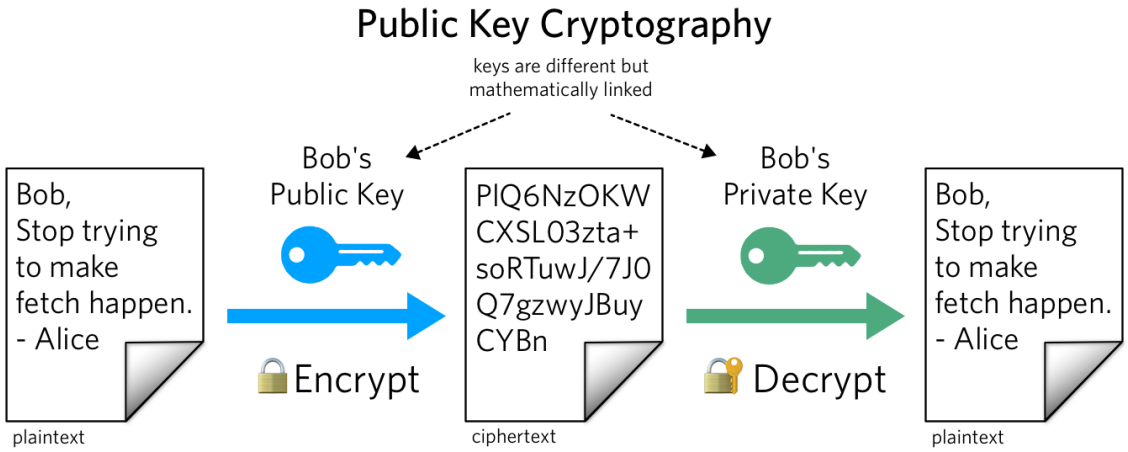


Figure 24

Encryption is the process of converting plaintext, the original readable data, into ciphertext, which is scrambled and unreadable without the decryption key. This process ensures that sensitive information remains secure from unauthorized access during transmission or storage.

Decryption is the reverse process of encryption, where the ciphertext is transformed back into plaintext, using a key, allowing the intended recipient to read the original message.

[24] https://www.twilio.com/en-us/blog/what-is-public-key-cryptography

8.2.1 Public and Private Cryptography

- **Symmetric Key Cryptography**: Uses the same key for both encryption and decryption. This key must be kept secret and shared between the sender and recipient beforehand.
- **Asymmetric Key Cryptography**: Utilizes a pair of keys – a public key, which can be shared widely, and a private key, which is kept secret by the owner. The public key encrypts messages, and the corresponding private key decrypts them.

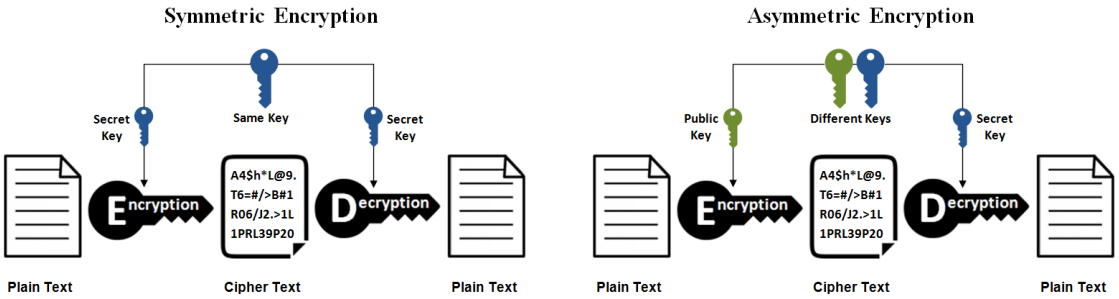


Figure 25

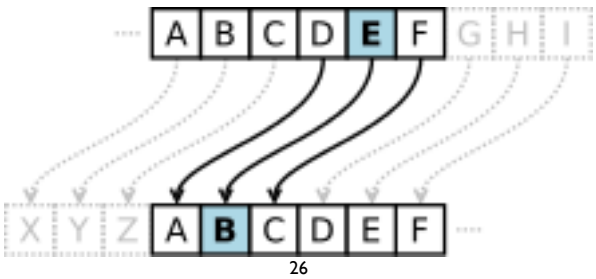
8.2.2 Performing Decryption

Decryption involves using the appropriate key to revert the ciphertext back to plaintext. The key used depends on the encryption method:

- In **symmetric** systems, the same key is used. The key's security is crucial as any leak compromises the data's security.
- In **asymmetric** systems, the private key must be securely stored as it is the only means to decrypt messages encrypted with the corresponding public key.

8.2.3 Mathematical Encryption and Decryption

An example of a simple mathematical approach to encryption is the Caesar cipher, which shifts the letters of the plaintext by a set number of positions. Although not secure, it demonstrates the basic concept of encryption.



[26]

8.3 Issues in Cryptographic Techniques

Cryptographic systems face several challenges:

---

[25] https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences
[26] https://en.wikipedia.org/wiki/Caesar_cipher

- **Key Distribution and Management**: Securely distributing and managing keys, especially in symmetric key cryptography, is complex and vulnerable to attack.
- **Cipher Weaknesses**: Some encryption methods can be vulnerable to types of cryptanalysis based on their structure or implementation errors.
- **Issues with Randomness**: Cryptography relies on randomness for generating keys. Predictability in this process can make encryption vulnerable to attacks.

8.4 Advanced Cryptographic Techniques

- **RSA Algorithm**: A widely used asymmetric encryption technique that relies on the factoring hardness of large integers. It is used for secure data transmission.
- **Advanced Encryption Standard (AES)**: A symmetric key algorithm that is widely used to secure data. AES is robust against various attacks and is efficient in both hardware and software.
- **Electronic Code Book (ECB)**: This is a mode of operation for block ciphers. It divides plaintext into blocks, and each block is encrypted separately. This simplicity can lead to patterns in the ciphertext, which is a potential weakness.

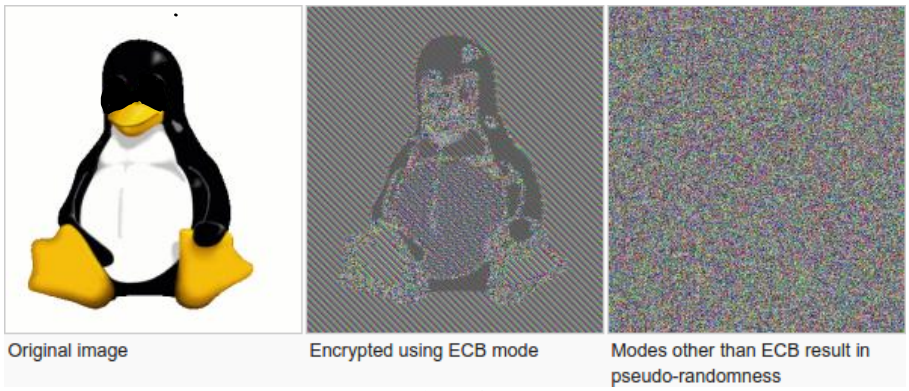


Figure 27

9 Network Security Concepts

Network security is essential to protect data during its transmission across networks. It encompasses various strategies and technologies to counter threats and vulnerabilities in network infrastructure.

9.1 TCP/IP Networking Architecture

TCP/IP (Transmission Control Protocol/Internet Protocol) is the foundational suite of communications protocols used on the Internet and similar computer networks. It is structured into four abstraction layers, which are the link layer, the internet layer, the transport layer, and the application layer.

[27] https://www.educative.io/answers/what-is-ecb

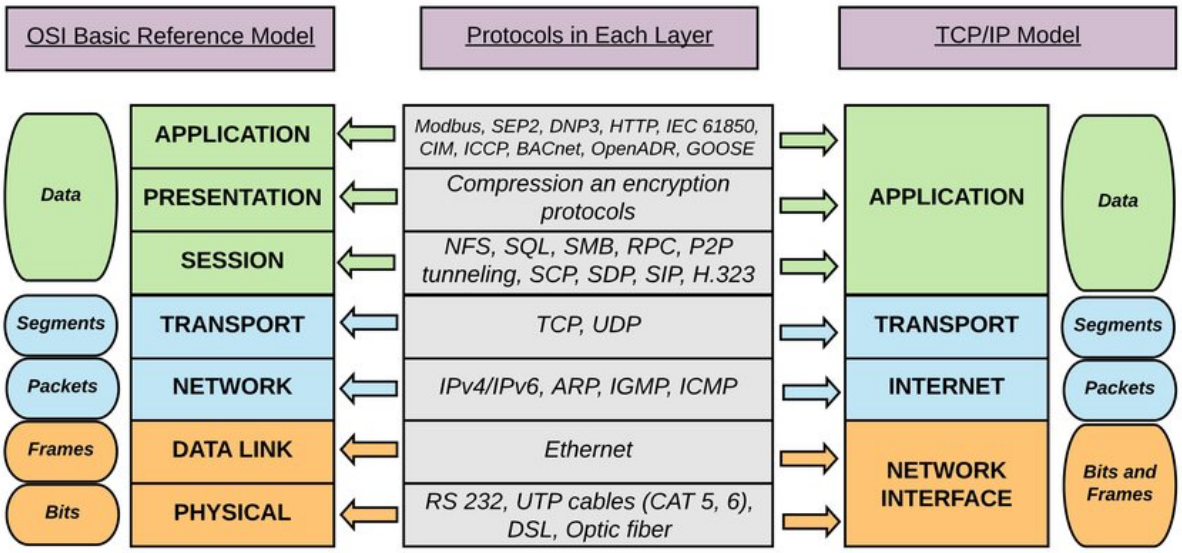


Figure 28

9.2 Common Networking Threats

- **IP Spoofing**: Where an attacker deceives a system by sending messages with a forged IP address.
- **SYN Flood**: A form of Denial-of-Service attack in which the attacker sends a succession of SYN requests to a target's system to consume enough server resources to make the system unresponsive to legitimate traffic.
- **Packet Sniffing**: Unauthorized interception and analysis of packets during their transmission.

9.3 Network Defences

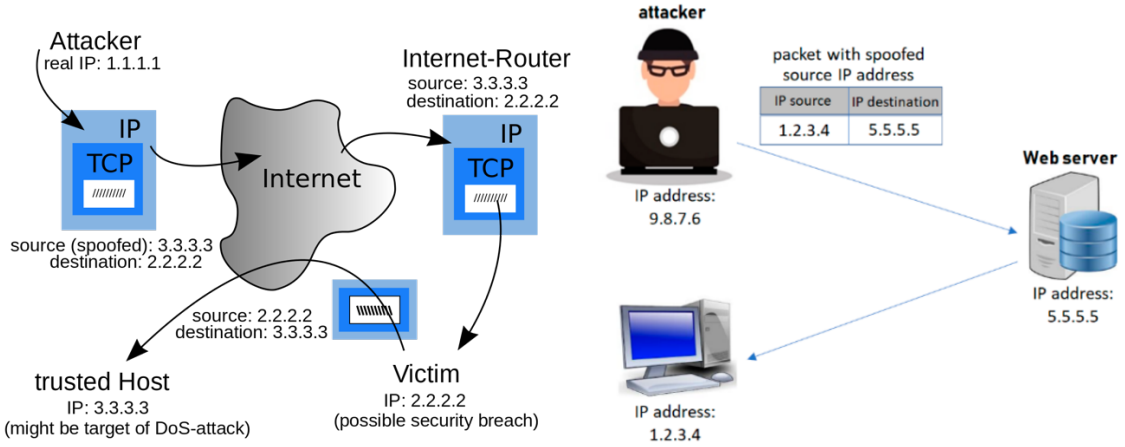


Figure 29, 30 (IP Spoofing)

[28] https://www.researchgate.net/figure/The-logical-mapping-between-OSI-basic-reference-model-and-the-TCP-IP-stack_fig2_327483011

[29] https://en.wikipedia.org/wiki/IP_address_spoofing

[30] https://www.mdpi.com/2073-431X/8/4/81

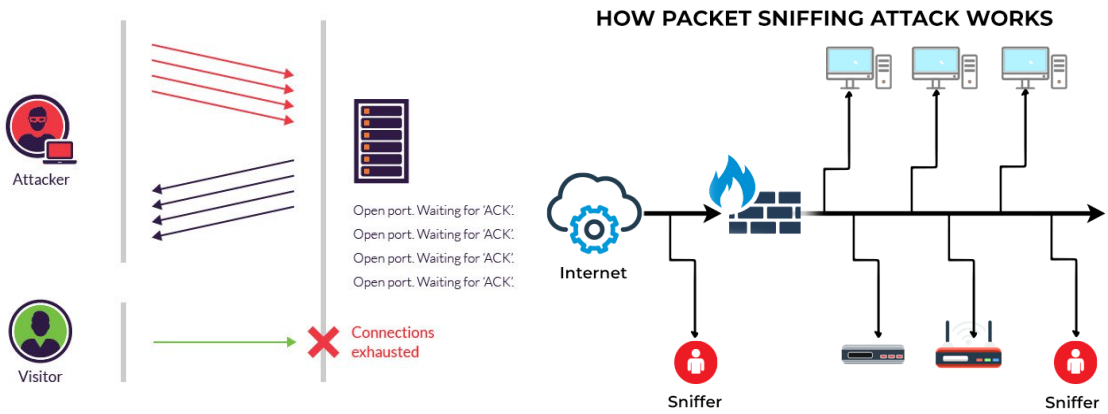


Figure 31 (SYN Flood), 32

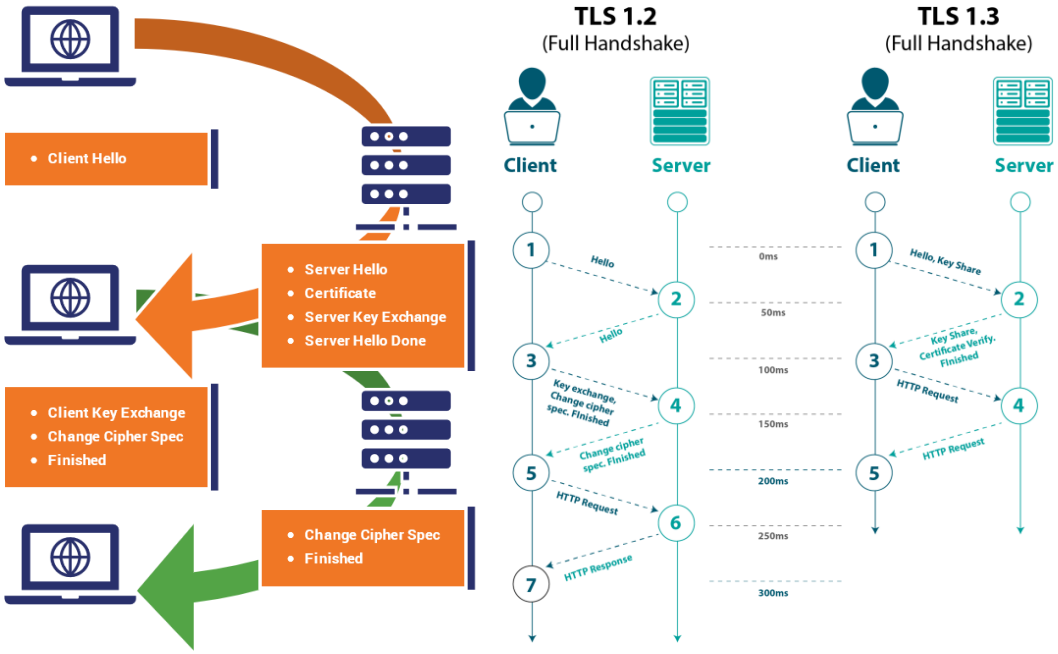


Figure 33, 34

9.3.2 Firewalls

A firewall is a network security device that monitors incoming and outgoing network traffic and decides whether to allow or block specific traffic based on a defined set of security rules. It's one of the first lines of defence in a network security system.

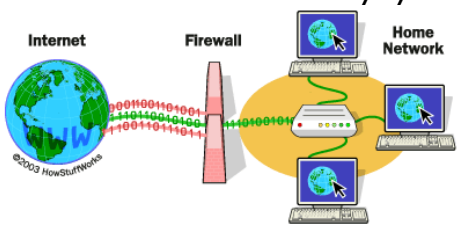


Figure 35

[31] https://www.imperva.com/learn/ddos/syn-flood/
[32] https://www.spiceworks.com/it-security/network-security/articles/what-is-packet-sniffing/
[33] https://www.thesslstore.com/blog/tls-1-3-everything-possibly-needed-know/
[34] https://www.appviewx.com/blogs/why-is-tls-1-3-better-and-safer-than-tls-1-2/
[35] https://www.comodo.com/resources/home/how-firewalls-work.php

9.3.3 Packet Filtering

Packet filtering is a firewall technique used to control network access by monitoring outgoing and incoming packets and allowing them to pass or halt based on the source and destination Internet Protocol (IP) addresses, protocols, and ports.

9.2.4 Defence Against Man-in-the-Middle Attacks

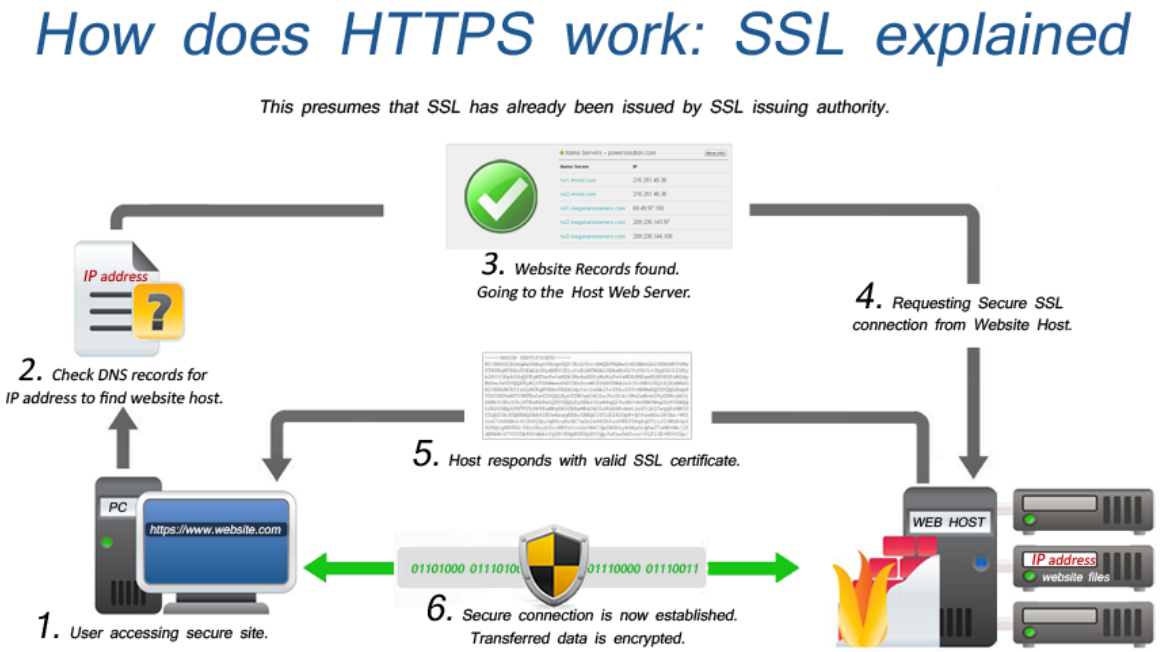


Figure 36

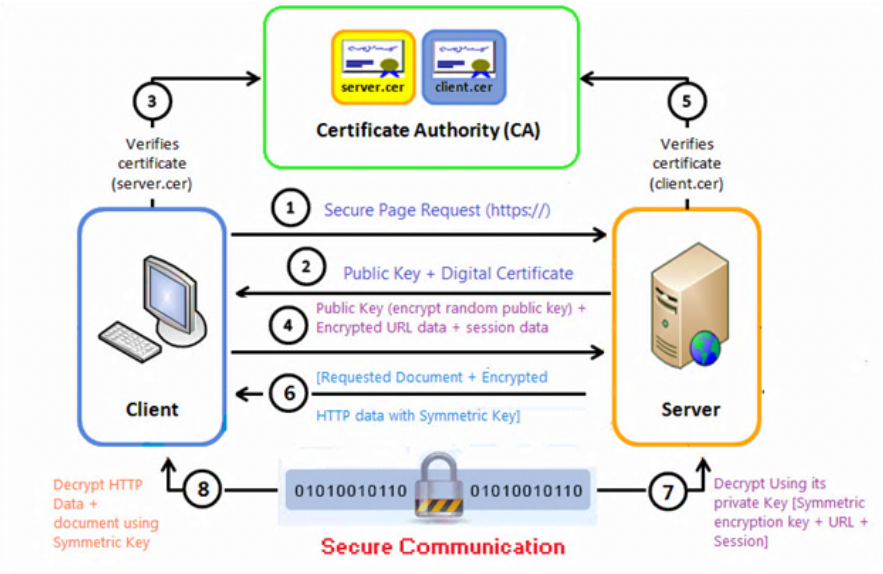


Figure 37

---

[36] https://mulakihost.com/tutorial/what-is-ssl-and-why-is-it-important

[37] https://superuser.com/questions/620121/what-is-the-difference-between-a-certificate-and-a-key-with-respect-to-ssl

Man-in-the-middle (MITM) attacks are where an attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other. Defences against MITM attacks include:

- **Public Key Infrastructure (PKI):** Uses certificates and public key cryptography to verify the identity of parties involved in communication.
- **SSL/TLS**: Secure Socket Layer and Transport Layer Security protocols that encrypt data in transit, preventing attackers from being able to read or modify the data.

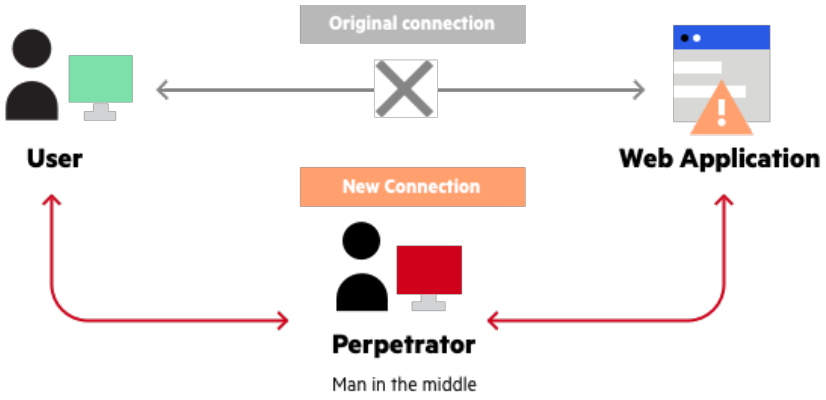


Figure 38

9.2.5 Domain Name Systems (DNS)

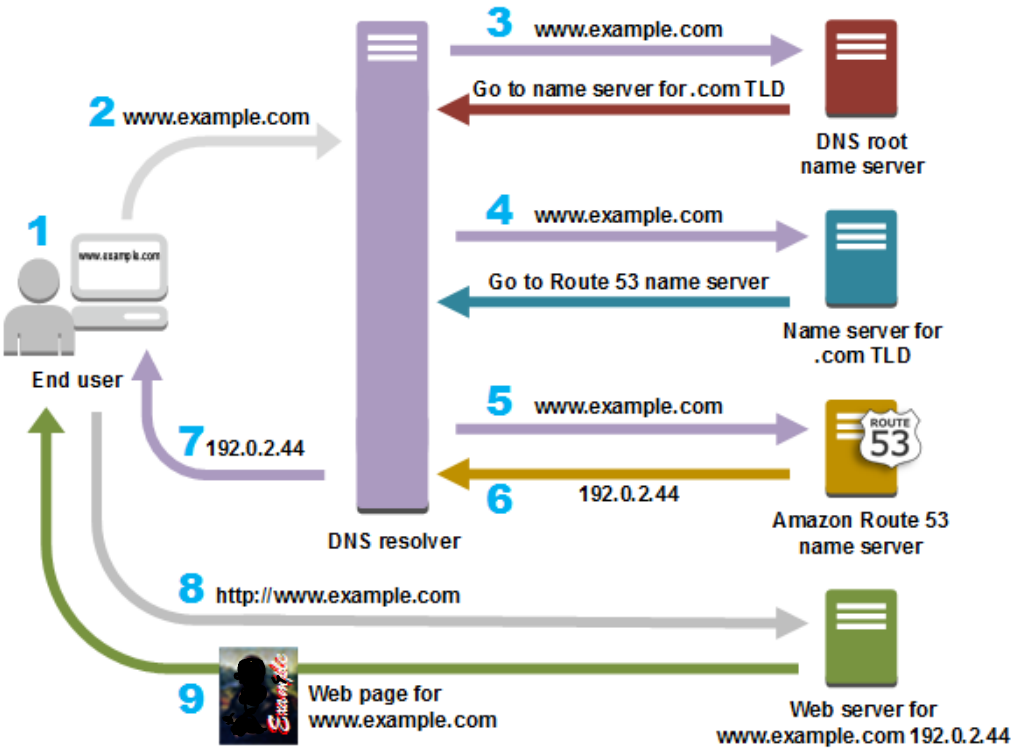


Figure 39

[38] https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/
[39] https://aws.amazon.com/route53/what-is-dns/

DNS Security: Domain Name System (DNS) translates human-readable domain names (like www.example.com) into machine-readable IP addresses. DNS security measures are critical to prevent DNS spoofing where attackers can redirect traffic from legitimate servers to fake ones. DNSSEC (DNS Security Extensions) adds security provisions to the DNS.

9.3 Network Monitoring Tools

9.3.1 Wireshark

Wireshark is a free and open-source packet analyser. It is used for network troubleshooting, analysis, software and communications protocol development, and educational purposes. Wireshark lets users put network interfaces in promiscuous mode to capture all the packets visible on that interface.

Example of Using Wireshark:

- **Capture Filters**: Specify which packets will be captured and which will be discarded. Used to decrease the size of the capture file.
- **Display Filters**: Used to filter packets displayed in the user interface of Wireshark while keeping the original data intact.

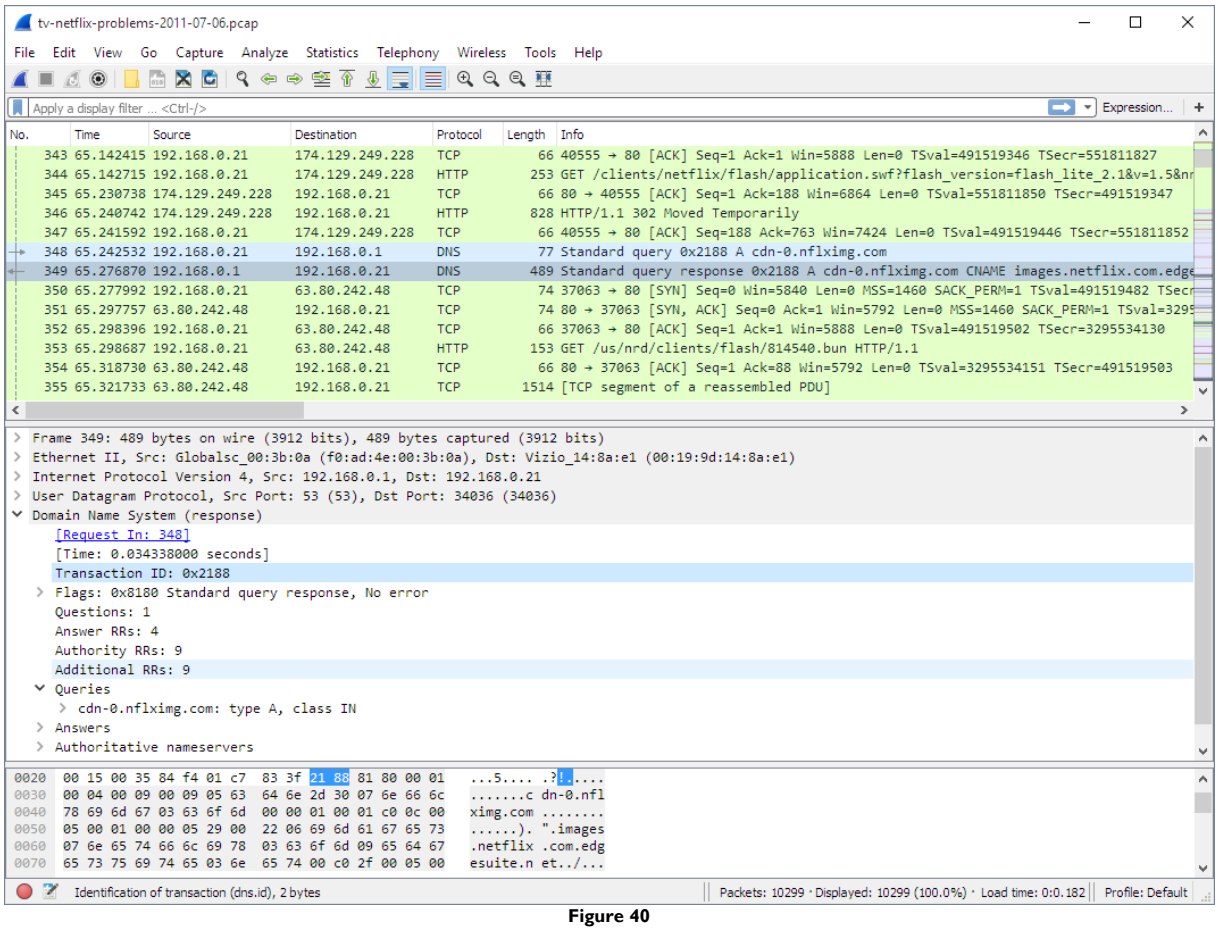


Figure 40

[40] https://www.wireshark.org/docs/wsug_html/

9.3.2 Cisco Packet Tracer

Cisco Packet Tracer is a powerful network simulation program that allows students to experiment with network behaviour and ask "what if" questions. It is particularly useful for educational institutions to teach and demonstrate complex technical concepts and networking systems design.

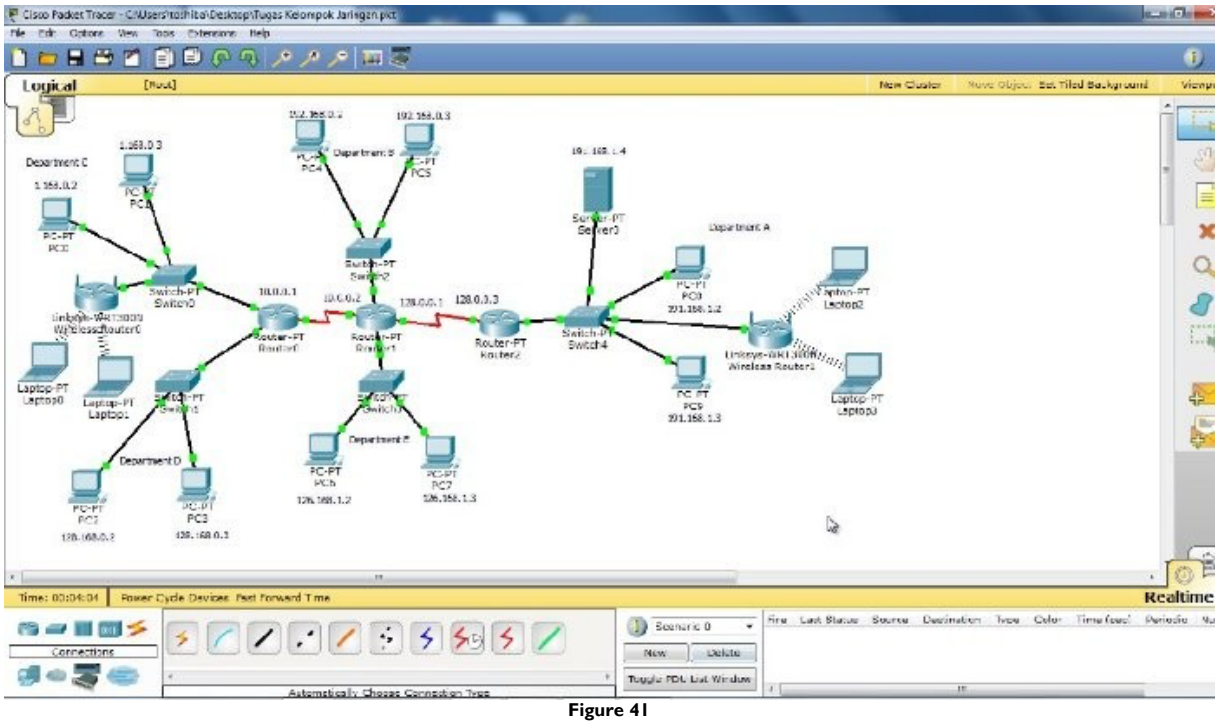


Figure 41

Features of Cisco Packet Tracer:

- **Simulation Mode**: Allows the user to view and control time intervals. The user can see the inner workings of data transfer across a network and observe how data is encapsulated and communicated.
- **Visualization Tools**: Helps in understanding the routing paths and networking processes by visualizing them graphically.

10 Cybersecurity Legislation and Regulation

Cybersecurity legislation varies significantly across different regions, including major economies and emerging markets. Each region has developed its laws and regulations based on its specific legal, social, and economic contexts and cyber threat landscape. Cybersecurity legislation and regulation are crucial in defining legal frameworks to protect data and manage the responsibilities of entities involved in digital environments.

Organisations that handle sensitive or personal data must comply with cybersecurity regulations. This compliance is about avoiding fines, protecting stakeholders, and maintaining trust in the digital economy.

[41] https://www.linkedin.com/pulse/ciscos-packet-tracer-now-available-free-paul-christian/

Cybersecurity ethics involves the study of ethical issues that arise about cyber technologies and environments. It includes considerations about privacy, data protection, and the morality of certain cybersecurity measures.

Ethical hacking involves hacking into a computer network to test and evaluate its security. The key here is that the hackers have permission to breach the network to find and fix security vulnerabilities.

10.1 Saudi Arabia

In Saudi Arabia, cybersecurity is governed by multiple legal frameworks and authorities, with the National Cybersecurity Authority (NCA) playing a leading role. Key regulations include:

- **Anti-Cyber Crime Law**: This law defines cybercrimes and the corresponding penalties which can include imprisonment and fines. It covers illegal access to computers, unauthorized access to bank data or any other personal data, and the dissemination of malware.
- **Personal Data Protection Law (PDPL)**: Recently enacted to regulate the processing of personal data by both public and private entities, enhancing data privacy and security.

10.2 United Kingdom

The UK has robust cybersecurity regulations, especially in response to the European Union's directives and the evolving digital landscape:

- **Data Protection Act 2018**: Supplements the EU's GDPR and tailors it to the UK context, controlling how personal information is used by organizations, businesses, or the government.
- **Network and Information Systems (NIS) Regulations 2018**: Aimed at improving national cybersecurity by mandating that organizations in critical sectors take measures to manage security risks and report major incidents.

10.3 United States

The United States addresses cybersecurity through a variety of federal laws and sector-specific regulations:

- **Cybersecurity Information Sharing Act (CISA) 2015**: Encourages the sharing of cybersecurity threat information between the government and companies.
- **Health Insurance Portability and Accountability Act (HIPAA)**: Protects the privacy of individual health information, mandating security provisions for protecting electronically protected health information.

States within the U.S. also have their cybersecurity regulations, like the California Consumer Privacy Act (CCPA), which offers broad privacy rights to residents.

10.4 European Union

The EU has one of the most stringent data protection and cybersecurity regimes:

General Data Protection Regulation (GDPR): A comprehensive data protection law that imposes obligations on all organizations that target or collect data related to people in the EU. Directive on Security of Network and Information Systems (NIS Directive): Requires member states to be better prepared for cyber-attacks, improving national cybersecurity capabilities and cross-border collaboration.

11 Hacking and Cyber Attacks

11.1 Cyber Attacks

Cyber-attacks are attempts by hackers to damage, steal data, or disrupt digital life in general. These attacks can target individuals, networks, and devices and can take various forms depending on the attacker's objectives and the vulnerabilities they exploit.

11.1.1 Brute Force Attacks

A brute force attack is a trial-and-error method used by application programs to decode encrypted data such as passwords or Data Encryption Standard (DES) keys, through exhaustive effort rather than employing intellectual strategies. This attack is simple and reliable; the attacker keeps trying to guess the password or key until they succeed. Key factors include:

Figure 42

- **Password Complexity**: Simple passwords can be guessed more easily and quickly.
- **Rate Limiting**: Systems without delays or lockouts after failed attempts are more vulnerable.

11.1.2 DDoS Attacks

Distributed Denial of Service (DDoS) attacks involve overwhelming a website or service with more traffic than the server or network can accommodate. The result is a denial of service for users of the targeted resource. Key aspects include:

- **Volume**: The attack sends a massive amount of traffic to the network.
- **Vectors**: It can come from many different sources, making it difficult to stop without affecting normal business operations.

[42] https://www.ssl2buy.com/wiki/brute-force-attack

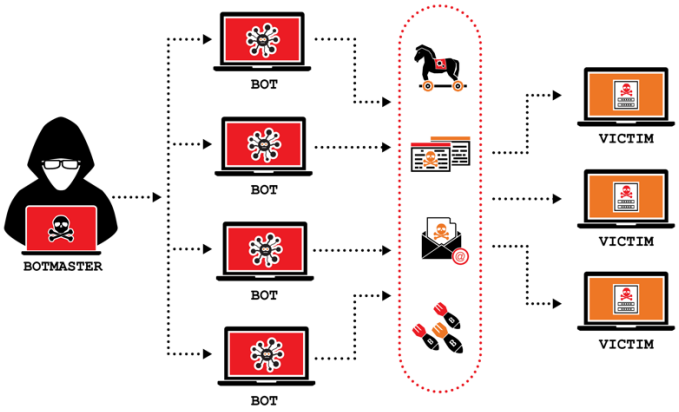


Figure 43

11.1.3 Other Attacks

Understanding the different types of cyber-attacks is crucial for developing effective cybersecurity strategies. Common attacks include phishing, malware, ransomware, DDoS (Distributed Denial of Service), and brute force attacks.

- **Buffer overflows** occur when more data is written to a buffer than it can hold. This can allow attackers to overwrite adjacent memory locations and execute malicious code.
- **SQL injection** is a code injection technique that might destroy your database. It is one of the most common web hacking techniques, placing malicious code in SQL statements via web page input.

11.2 Malware and Viruses

Malware, short for malicious software, encompasses various types of harmful software designed to disrupt, damage, or gain unauthorized access to a computer system. Viruses, a type of malware, are programs that can replicate themselves and spread to other devices, often corrupting data or taking control of system resources.

11.2.1 Logic Bombs

Logic bombs are malicious programs that execute a harmful function when specified conditions are met. These conditions can be based on the passage of time or the occurrence of a specific event, such as the deletion of an employee's credentials from a company database.

They deliver the payload after fulfilling a certain requirement: The prerequisite is the detonator for the logic bomb. This characteristic enables logic code bombs to remain undetected for extended periods of time. The date of a significant event or the deletion of an employee from the firm's payroll could serve as the trigger. **Time bombs** are also present in logic bombs, with triggers connected to dates or specific times.

They have an unknown payload: This is true until the payload is activated. The part of malware known as the payload is responsible for carrying out the harmful action; in other words, it determines what kind of harm the virus is designed to cause. The payload may cause the theft of sensitive data or the distribution of **spam emails** through an affected system.

They are latent until they are activated: Logic bombs are not intended to explode immediately, much like a ticking time bomb. To hide their identities, **attackers that target a system from within frequently utilize logic bombs.** Subtle logic bombs can go unnoticed for years.

Figure 44

[43] https://www.thesslstore.com/blog/what-is-a-ddos-attack/
[44] https://www.zenarmor.com/docs/network-security-tutorials/what-is-logic-bomb

11.2.2 Backdoors

Backdoors are secretive entry points into a computer system or program that bypass normal authentication procedures. They allow attackers to gain remote access to a device or network, often without the knowledge of the system's owner.

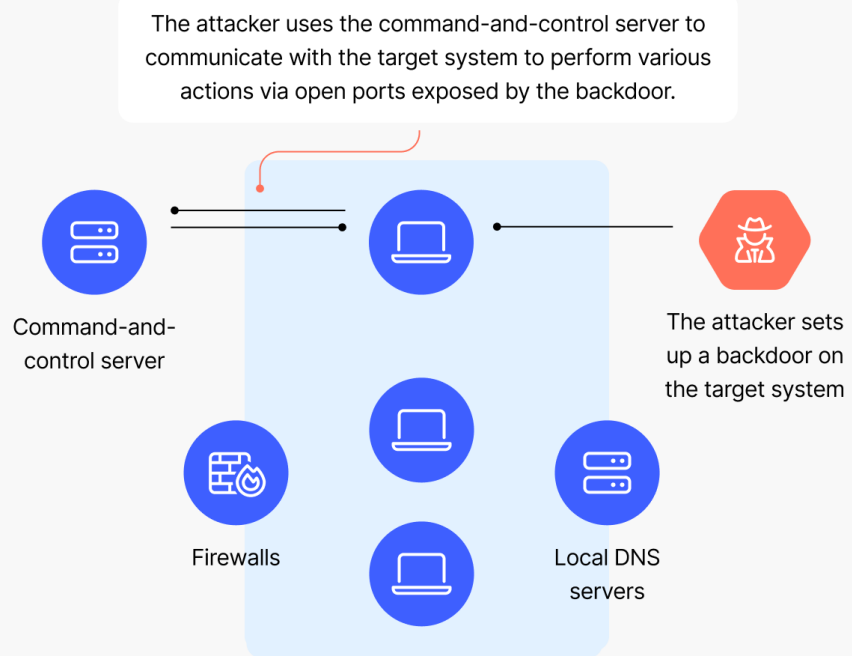


Figure 45

11.2.3 Viruses

Viruses are malicious software programs that, when executed, replicate by modifying other computer programs and inserting their own code. Once this replication succeeds, the affected areas are then said to be "infected." Viruses can spread through many means, including executable files, scripts, or documents.

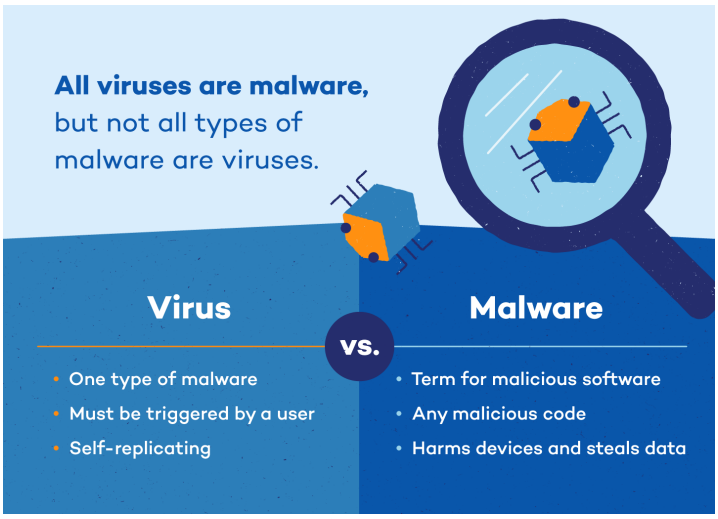


Figure 46

[45] https://nordvpn.com/blog/backdoor-attack/
[46] https://www.xcitium.com/malware-vs-virus/

11.2.4 Supply Chain Attacks

Supply chain attacks involve malicious interference with the production or distribution process of software to insert malicious code into legitimate software packages. This type of attack targets less-secure elements in the supply network to gain access to more secure targets.

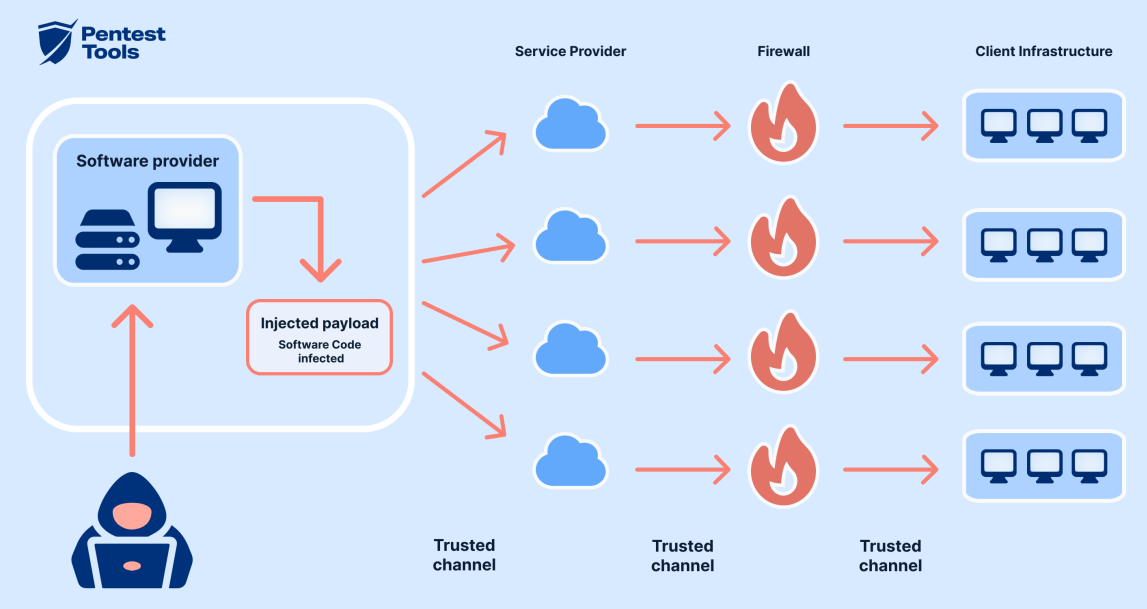


Figure 47

11.2.5 Trojan Horses

Trojan horses are a type of malicious code or software that tricks users into running it by masquerading as legitimate software. Trojans can perform a variety of functions (like creating backdoors) but do not replicate themselves.

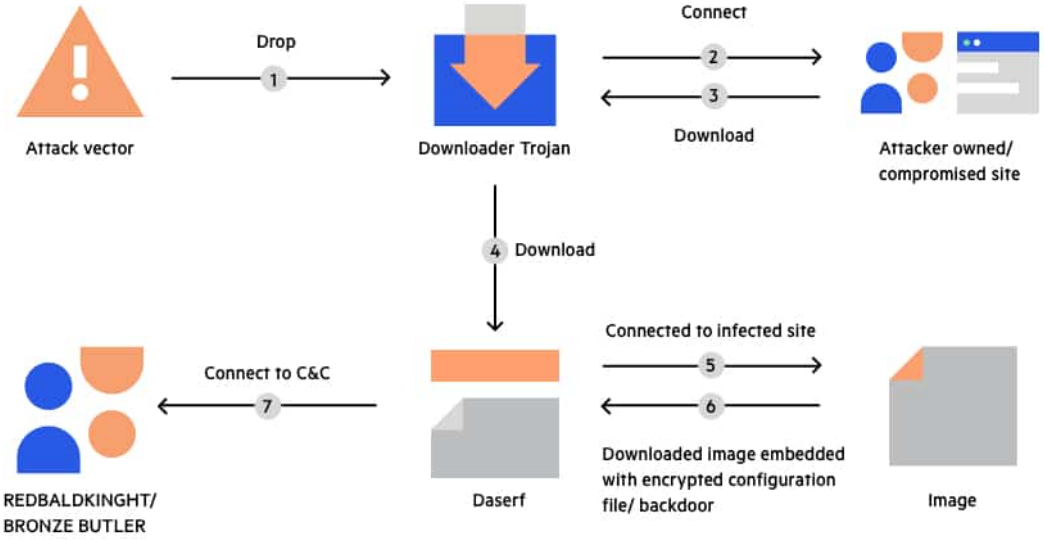


Figure 48

[47] https://pentest-tools.com/blog/supply-chain-attacks
[48] https://www.imperva.com/learn/application-security/trojans/

11.2.6 Worms
Worms are malware computer programs that replicate themselves to spread to other computers, often over a computer network. Unlike viruses, worms do not need to attach themselves to an existing program.

11.2.7 Methods to Prevent Malware and Viruses
Effective strategies to combat malware include using antivirus software, keeping systems updated, and educating users about secure practices. These measures help in identifying threats, isolating harmful software, and removing it from the system.

Antivirus software is critical in the fight against malware and viruses. It scans the computer system for known threats and can often predict new ones through heuristic evaluation and signature databases. Key functions include real-time scanning, periodic updates, and the quarantine of suspicious files.

Command injection is a security vulnerability that allows an attacker to execute arbitrary commands on a host operating system via a vulnerable application. Prevention techniques include:

- **Validating Input**: Ensuring that all user input is validated before processing to prevent malicious data from being used in command execution.
- **Using Safe APIs**: Utilizing APIs that avoid the use of the operating system shell when calling system commands, thereby reducing potential attack vectors.

11.2.8 Input Validation
Input validation is a method by which input supplied by the user is validated to ensure it conforms to application requirements before being processed. It helps in preventing not just SQL injection and cross-site scripting (XSS) but also command injection attacks.

11.3 Threat-Vectors and Threat Agent

Threat vectors are the methods or pathways through which cybercriminals launch attacks on networks, computers, and other resources. A deeper understanding of these vectors helps in developing more sophisticated and effective defence strategies against potential threats.

Firewalls act as barriers that monitor and control incoming and outgoing network traffic based on predetermined security rules. Intrusion Detection Systems (IDS) are devices or software applications that monitor networks or systems for malicious activity or policy violations.

Secure network practices involve comprehensive strategies to protect data during transmission, ensure the integrity of network infrastructure, and maintain the confidentiality of information.

11.3.1 Email and Phishing Attacks
One of the most common threat vectors is email. Cybercriminals use phishing tactics to trick users into providing sensitive information, clicking on malicious links, or opening attachments that contain malware. Phishing emails may appear to be from legitimate sources, often mimicking the look and feel of emails from trusted institutions or contacts.

Prevention Techniques:

- **Educating users** about the risks of unsolicited emails.
- **Implementing email filters** that detect phishing attempts.
- Using **multi-factor authentication (MFA)** to add an additional layer of security.

11.3.2 Web-Based Threats

Web-based threats involve malicious software or content hosted on websites which may exploit browser vulnerabilities to execute harmful scripts or download malware onto a user's device. Drive-by downloads and malvertising are common types of web-based threats.

Prevention Techniques:

- Keeping browsers and plugins **updated** to mitigate known vulnerabilities.
- Using **web filters** to block access to malicious websites.
- Employing **browser security settings** to restrict unauthorized downloads and plugins.

11.3.3 Malicious Insider Threats

Insiders, such as employees or contractors, who have legitimate access to an organization's network, can become a threat vector if they misuse their access to steal information, sabotage systems, or introduce malware.

Prevention Techniques:

- Implementing **strict access controls** and using the principle of least privilege.
- **Monitoring and auditing user activities** to detect and respond to suspicious behaviour promptly.
- Conducting **regular security awareness** training to deter unintentional insider threats.

11.3.4 Removable Media

USB drives, external hard drives, and other forms of removable media can serve as a direct line for viruses and malware to bypass network security measures and infect a system.

Prevention Techniques:

- **Disabling auto-run features** on all systems to prevent automatic execution of malicious software.
- **Restricting the use of removable media** to only approved devices.
- **Scanning all removable media** with antivirus software before use.

11.3.5 Wireless Networks

Insecure wireless networks are vulnerable to eavesdropping and attacks such as man-in-the-middle (MITM). Attackers can intercept or manipulate the data transmitted over these networks without physically connecting to them.

Prevention Techniques:

- Using **strong, complex passwords** for network access.
- Implementing **strong encryption methods** like WPA3 for wireless communications.
- Securing network equipment by **changing default configurations and disabling WPS**.

11.3.6 Supply Chain Attacks

Supply chain attacks target less-secure elements in the supply network to compromise the security of major organizations. This can involve compromising hardware or software vendors or third-party service providers.

Prevention Techniques:

- Conducting thorough **security audits** of all suppliers and third-party vendors.
- Implementing **robust incident response plans** that include third-party risks.
- Ensuring that suppliers adhere to **security best practices** and standards

This concludes the Course Reader for the core topics for the Oxmedica Cybersecurity and Cryptography Course. There will be another provided for the specialist topics إن شاء الله.