MODULE 3.1

Encryption / Decryption with Python

Program

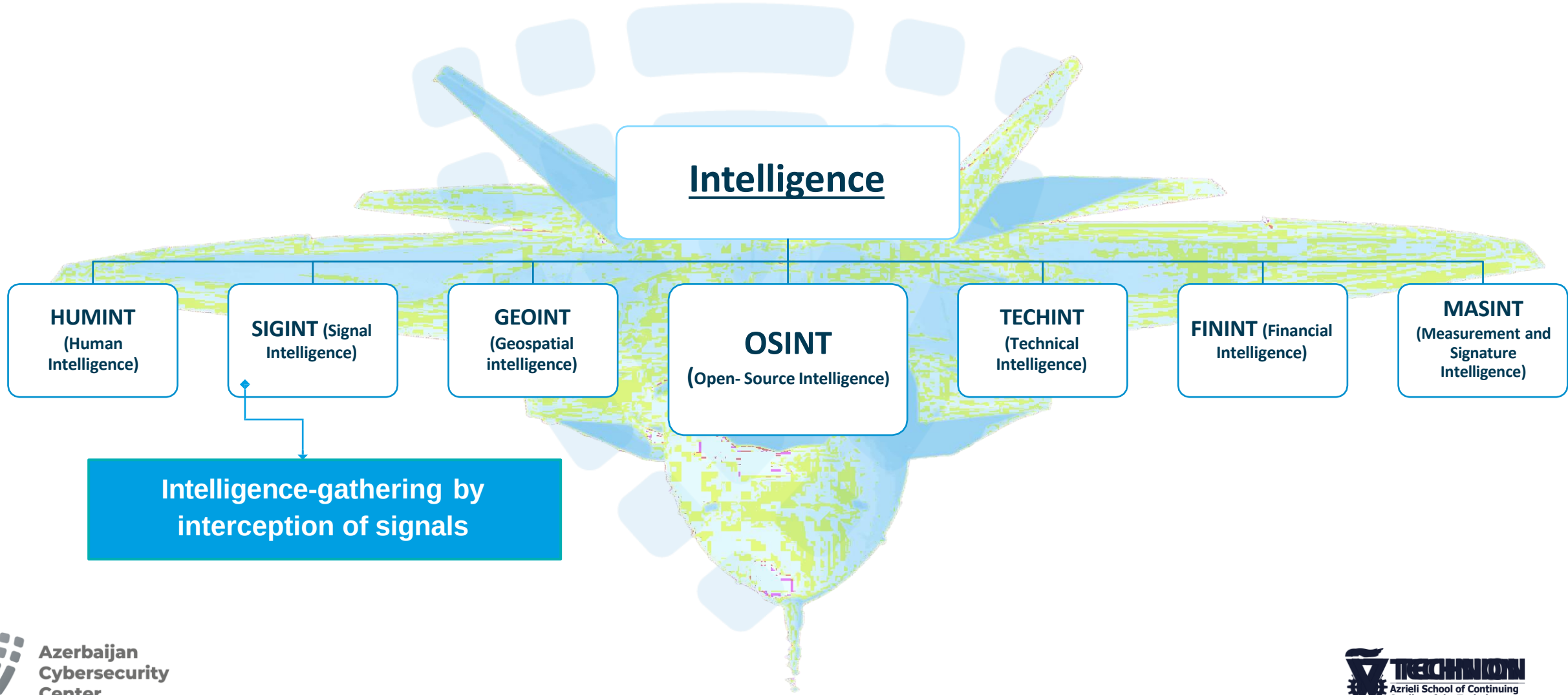
- Introduction to Encryption / Decryption
- ASCII Encoding / Decoding
- **Lab 3.1 Breaking the Caesar cipher**
- **Lab 3.2 Breaking the transposition cipher**
- Working with files
- Asymmetric encryption: The RSA algorithm
- Cryptography libraries in Python

Learning Objectives

- You will be able **to program** simple cipher algorithms, like the Caesar cipher and the transposition cipher
- You will be **able to read** and **to write** files to the file system with Python
- You will be able **to break** substitution ciphers by frequency analysis
- You will be able **to use** the RSA cryptography library in Python
- You will be able **to understand** timing attacks against python login inputs

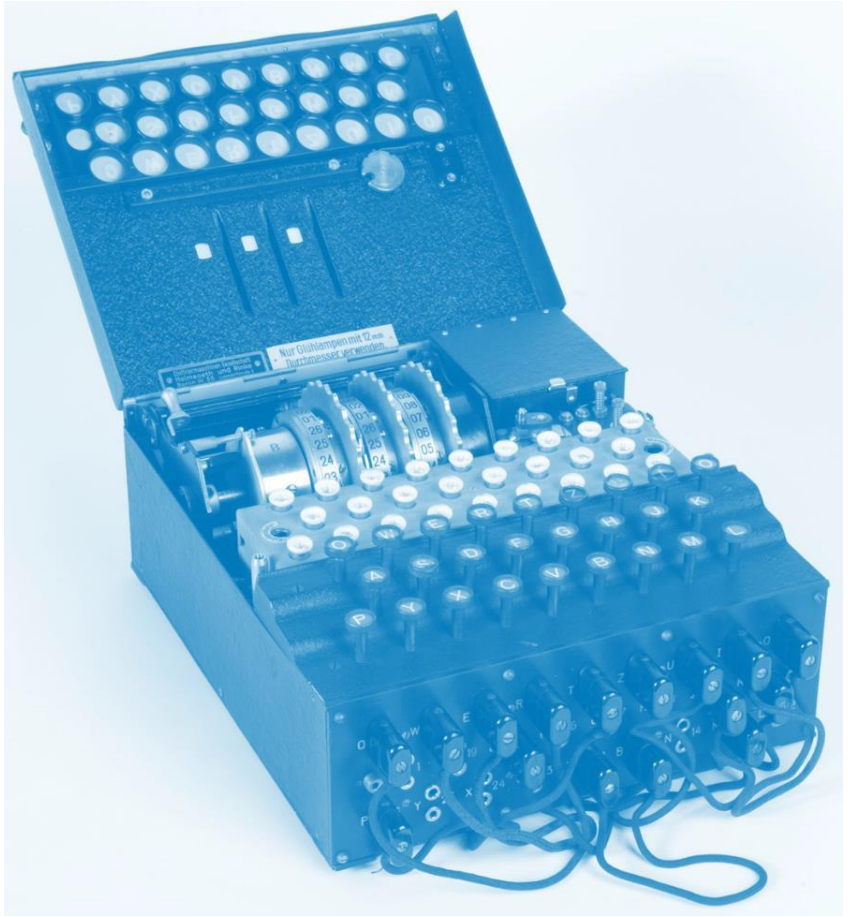
Introduction to Encryption / Decryption

Intelligence Gathering Disciplines



SIGINT

- The **Enigma machine** was a cipher machine used during World War II for encrypting and decrypting secret messages.
- The Enigma machine was a symmetric encryption device that used a combination of substitution and permutation (transposition) techniques.
- The breaking of the Enigma cipher by the Allies had a profound impact on the outcome of the war, as it enabled them to intercept and decrypt sensitive German communications.



Cipher Methods

Cipher Name	Characteristics	Type
Caesar Cipher	- Substitution cipher where each letter is shifted by a fixed number of positions in the alphabet. Only has 26 possible keys (1 for each shift value).	Symmetric
Transposition Cipher	- Reorganizes the order of letters in the plaintext, without changing the actual letters themselves. Does not substitute or replace letters.	Symmetric
Affine Cipher	- Combines both substitution and linear transformation. - Each letter is replaced by a mathematical formula (ax + b) mod 26, where a and b are fixed coefficients.	Symmetric
Substitution Cipher	- Each letter in the plaintext is replaced with another letter or symbol according to a fixed substitution rule. - Simplest form is the Caesar cipher.	Symmetric
Vigenère Cipher	- Extension of the Caesar cipher where each letter is shifted by a value from a keyword, repeating the keyword as necessary. - Provides stronger encryption compared to the Caesar cipher.	Symmetric
One-Time Pad Cipher	- Uses a random key that is as long as the plaintext, and the key is used only once. - Provides perfect secrecy when used correctly.	Symmetric
RSA Cipher	- Public-key encryption algorithm that uses a pair of keys: public key for encryption and private key for decryption. Based on the difficulty of factoring large numbers.	Asymmetric
Elliptic Curve Cipher	- Public-key encryption algorithm that uses points on an elliptic curve over a finite field for encryption and decryption. Provides similar security to RSA but with shorter keys.	Asymmetric

ASCII Encoding / Decoding

How Encodings Work?

- In order to create bytes, we need to **encode** strings.
- An **encoding** is a way to turn a string into a series of bits
- There are many types of encoding available…
- The most common character set is **ASCII** – in which all characters are encoded into 1byte (8 bits). These files are called *plaintext files*.

The ASCII Table

```
┌──(kali㉿kali)-[~]
└─$ ascii -a
Usage: ascii [-adxohv] [-t] [char-alias ... ]
   -t = one-line output  -a = vertical format
   -d = Decimal table  -o = octal table  -x = hex table  -b binary table
   -h = This help screen -v = version information
Prints all aliases of an ASCII character. Args may be chars, C \-escapes,
English names, ^-escapes, ASCII mnemonics, or numerics in decimal/octal/hex.

Dec Hex    Dec Hex    Dec Hex  Dec Hex  Dec Hex  Dec Hex   Dec Hex   Dec Hex
  0 00 NUL  16 10 DLE  32 20    48 30 0  64 40 @  80 50 P   96 60 `  112 70 p
  1 01 SOH  17 11 DC1  33 21 !  49 31 1  65 41 A  81 51 Q   97 61 a  113 71 q
  2 02 STX  18 12 DC2  34 22 "  50 32 2  66 42 B  82 52 R   98 62 b  114 72 r
  3 03 ETX  19 13 DC3  35 23 #  51 33 3  67 43 C  83 53 S   99 63 c  115 73 s
  4 04 EOT  20 14 DC4  36 24 $  52 34 4  68 44 D  84 54 T  100 64 d  116 74 t
  5 05 ENQ  21 15 NAK  37 25 %  53 35 5  69 45 E  85 55 U  101 65 e  117 75 u
  6 06 ACK  22 16 SYN  38 26 &  54 36 6  70 46 F  86 56 V  102 66 f  118 76 v
  7 07 BEL  23 17 ETB  39 27 '  55 37 7  71 47 G  87 57 W  103 67 g  119 77 w
  8 08 BS   24 18 CAN  40 28 (  56 38 8  72 48 H  88 58 X  104 68 h  120 78 x
  9 09 HT   25 19 EM   41 29 )  57 39 9  73 49 I  89 59 Y  105 69 i  121 79 y
 10 0A LF   26 1A SUB  42 2A *  58 3A :  74 4A J  90 5A Z  106 6A j  122 7A z
 11 0B VT   27 1B ESC  43 2B +  59 3B ;  75 4B K  91 5B [  107 6B k  123 7B {
 12 0C FF   28 1C FS   44 2C ,  60 3C <  76 4C L  92 5C \  108 6C l  124 7C |
 13 0D CR   29 1D GS   45 2D -  61 3D =  77 4D M  93 5D ]  109 6D m  125 7D }
 14 0E SO   30 1E RS   46 2E .  62 3E >  78 4E N  94 5E ^  110 6E n  126 7E ~
 15 0F SI   31 1F US   47 2F /  63 3F ?  79 4F O  95 5F _  111 6F o  127 7F DEL
```

Azerbaijan Cybersecurity Center

TECHNION Azrieli School of Continuing Studies of the Technion

What Is Encoding?

- To encode is to translate a string into the bytes that represent each letter.
- ASCII is the easiest encoding, useful only with English. We will learn other encodings in the future.
- For example, encoding ‘A’ using ASCII is to translate ‘A’ into the hexadecimal byte 41, written in Python as `‘\x41’`.
- This turns a `string` object into a `bytes` object.

Creating Bytes in Python

```
In [1]: 'A'.encode()
Out[1]: b'A'
```

- It's as easy as that.
- The ***b*** before the quotes shows that these are bytes, not a string.

```
In [2]: type(b'Hello, World!')
Out[2]: bytes
```

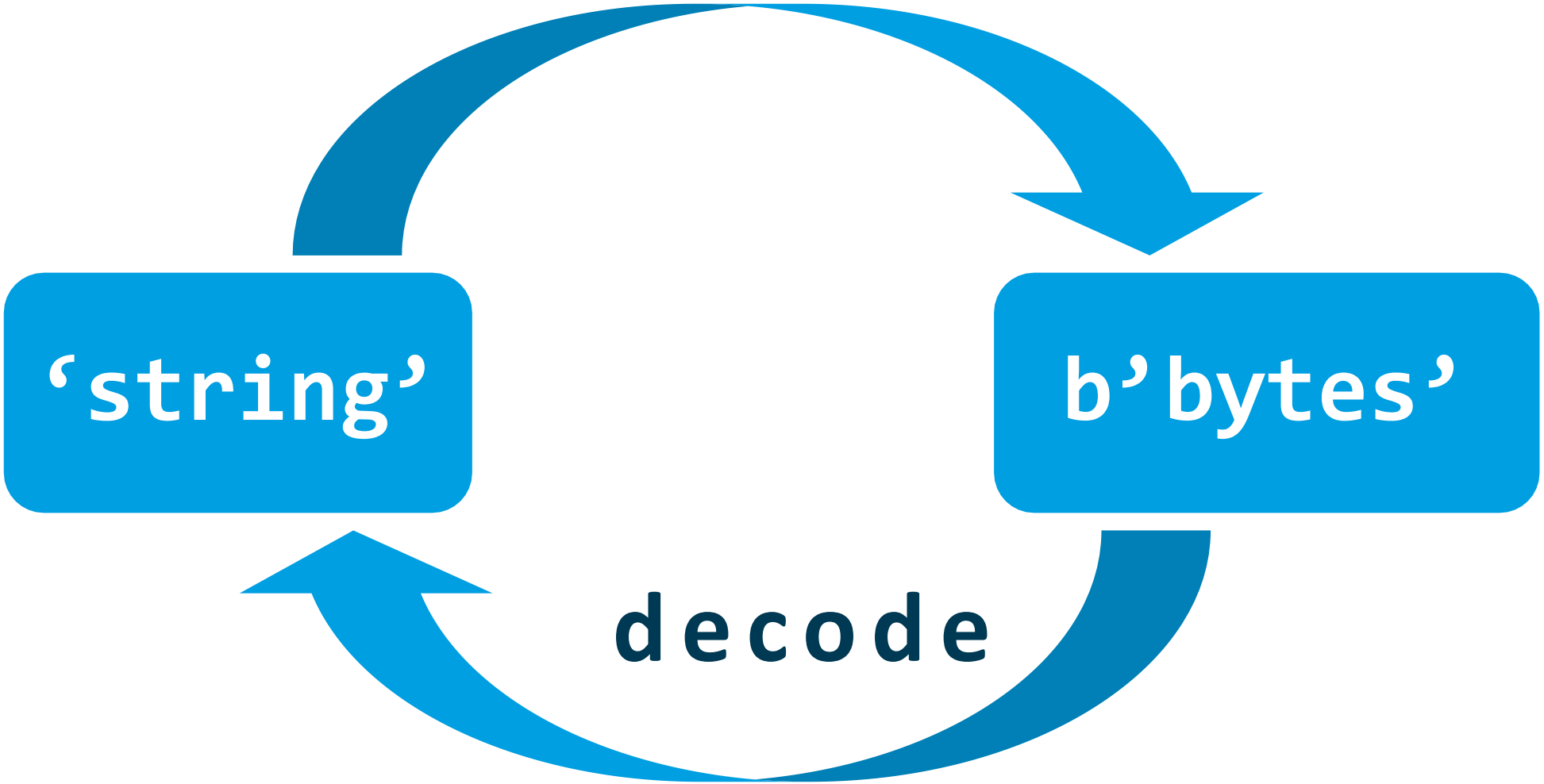
- This is what we can use in order to write strings to files!
- We can also create bytes by ourselves, by writing ***b*** before the string definition:

Converting Bytes into Strings

- The same can be done the other way around.
- Bytes can be converted into the strings that they represent by using the .decode() string method:

```
In [3]: alphabet = b'abcdefg'

In [4]: type(alphabet)
Out[4]: bytes

In [5]: alphabet = alphabet.decode()

In [6]: type(alphabet)
Out[6]: str
```

encode



The chr() and ord()functions

- **chr(ascii_value)** returns a string representing a character whose Unicode code point is the integer ascii_value. In other words, it takes an ASCII value (or Unicode code point) as input and returns the corresponding character as a string.
  - For example, chr(65) returns the string 'A', chr(97) returns the string 'a', and so on.
- **ord(character)** returns an integer representing the Unicode code point of the given character. In other words, it takes a character as input and returns its corresponding ASCII value (or Unicode code point) as an integer.
  - For example, ord('A') returns the integer 65, ord('a') returns the integer 97, and so on.

Lab 3.1
Breaking the Caesar Cipher

Lab 3.2
Breaking the Transposition Cipher

Working with files in Python

The os.path and Path modules

- Both os.path and Path modules provide similar functionalities for checking paths, file existence, and directory existence.
- However, there are some differences in the usage and syntax.

Function	os.path	Path (from pathlib)
Check a path	os.path.exists(path)	Path(path).exists()
Check if file exists	os.path.isfile(path)	Path(path).is_file()
Check if directory exists	os.path.isdir(path)	Path(path).is_dir()

Example with os.path

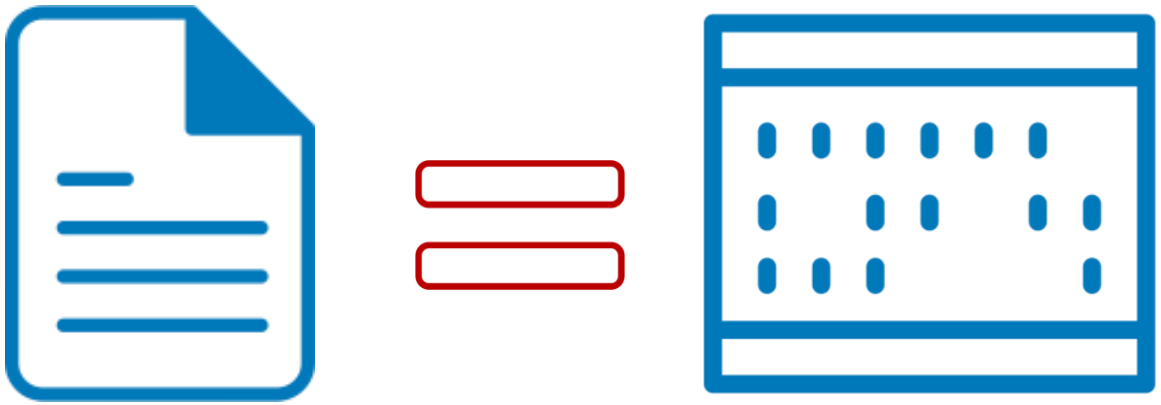
```
>>> import os
>>> os.path.exists("/home/kali/file-1")
True
>>> os.path.exists("/home/kali/file-2")
False
```

Example with Path

```
>>> from pathlib import Path
>>> Path.cwd()
PosixPath('/home/kali')
>>> my_path = Path('/home/kali/file-1')
>>> my_path.exists()
True
>>> my_path = Path('/home/kali/file-2')
>>> my_path.exists()
False
```

What Are Files?

- Files are a long sequence of bytes, saved on the hard disk.
  - Not characters - Not letters – **Bytes (= 8 Bits)**
  - Sometimes we will want to read them, and sometimes to write them.
  - Thankfully, this is very easy to do in Python!



Accessing a File

- To access a file, we will use the open function.
- **Open** receives two inputs: the file path, and the mode.
- We will use one of two modes:
  - “rb” – used for reading files
  - “wb” – used for writing files

Closing the File

- When we access a file and open it, our Operating System locks it only for us. This means no one else can access it while the file is open.
- This means that once we are done using the file (reading/writing), we will always want to close it. If we don't others will not be able to access it.
- We close the file using the .close() method.

Reading a File

- **Step 1**– Opening the file – using the open() function, and giving the path, and the mode ‘rb’
- **Step 2** – Reading the data – using the method .read()
- **Step 3** – Closing the file – using the method .close()

Reading a File (cont.)

```
In [7]: file = open(r"C:\python\hello.txt", 'rb')

In [8]: text_bytes = file.read()

In [9]: file.close()


In [10]: text_bytes
Out[10]: b'Hello, World!'

In [11]: text = text_bytes.decode()

In [12]: print(text)
Hello, World!
```

Writing a File (process)

- **Step 1**– Opening the file – using the open() function, and giving the path, and the mode 'wb'
- **Step 2** – Writing the data – using the method `.write()`, and giving the bytes we want to write.
- **Step 3** – Closing the file – using the method `.close()`

Writing a File (cont.)

```
In [21]: text = 'Agent Bell, reporting for duty!'

In [22]: text_bytes = my_text.encode()

In [23]: text_bytes
Out[23]: b'Agent Bell, reporting for duty!'

In [24]: file = open(r'C:\Python\agent.txt', 'wb')

In [25]: file.write(text_bytes)
Out[25]: 31

In [26]: file.close()
```

The Path of the File to Write

- The path you give the open() function when writing can be an existent file, or an inexistent file.
  - If the file doesn't exist – a new file will be created.
  - If the file exists – the existing file will be erased and replaced.
- But, if the directory the file is located in doesn't exist – an error will be raised.

Reading

```
In [1]: file = open(r'C:\Python\temp.txt', 'rb')

In [2]: data_bytes = file.read()

In [3]: file.close()

In [4]: data = data_bytes.decode()

In [5]: data
Out[5]: 'aaaa\r\nbbb\r\ncc\r\nd'

In [6]: print(data)
aaaa
bbb
cc
d
```

Splitting Text into Lines

- We can use the .split() method, with the newline character as the delimiter.
- We can also use the .splitlines() method, which is much easier!

```
In [7]: data_lines = data.split('\r\n')

In [8]: data_lines
Out[8]: ['aaaa', 'bbb', 'cc', 'd']

In [9]: data_lines = data.splitlines()

In [10]: data_lines
Out[10]: ['aaaa', 'bbb', 'cc', 'd']
```

Reading Lines from a File

```
In [11]: file = open(r'C:\Python\temp.txt', 'rb')

In [12]: data = file.read().decode()

In [13]: file.close()

In [14]: for line in data.splitlines():
    ...:     print(line)
    ...:
aaaa
bbb
cc
d
```

The .read() Method

- Read, when executed without an input, reads all of the bytes of the file. What happens if we use it twice?

```
In [15]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [16]: file.read()
Out[16]: b'abcdefghijklmnopqrstuvwxyz'

In [17]: file.read()
Out[17]: b''
```

File Pointer

- Just like in variables, our pointer shows us where in the file we read from.
- When the file has just been opened, the file pointer points to the beginning of the file. As the file is being read, the pointer advances over the bytes of the file.
- When all bytes have been read, the pointer is already at the end of the file, so executing the `.read()` method again will read 0 more bytes.

File Pointer in Action

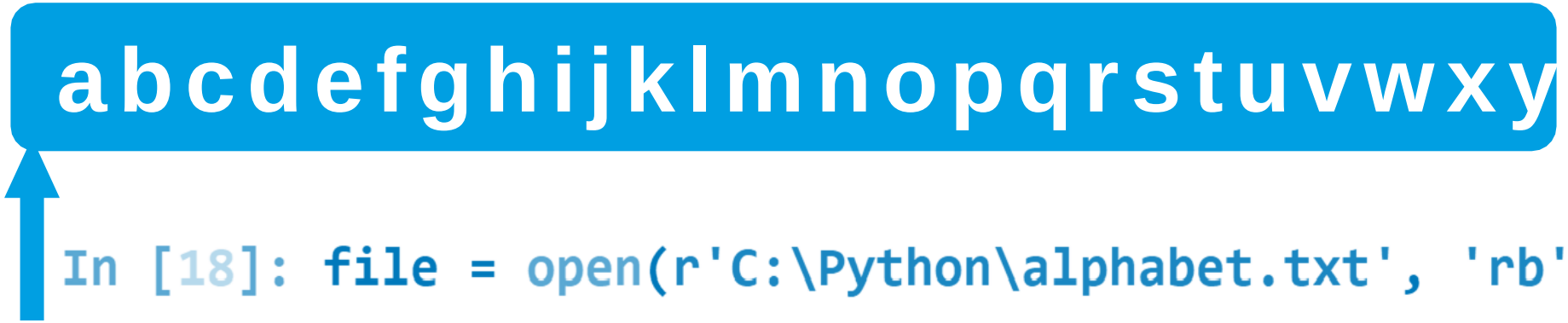

```
In [15]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [16]: file.read()
Out[16]: b'abcdefghijklmnopqrstuvwxyz'

In [17]: file.read()
Out[17]: b''
```

.read(num_of_bytes)

- Instead of using **.read()** to read the whole file, we can give it the number of bytes to read:

```
# Read the first 100 bytes from the file
data = file.read(100)
# Read the next 50 bytes from the file
data = file.read(50)
```

- **Note:** `file.iter_content()` reads the contents of a file in chunks as bytes objects, typically used for processing large files that may not fit into memory.




```
In [18]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [19]: file.read(2)
Out[19]: b'ab'

In [20]: file.read(4)
Out[20]: b'cdef'

In [21]: file.read(8)
Out[21]: b'ghijklmn'

In [22]: file.read()
Out[22]: b'opqrstuvwxyz'
```

Where is My Pointer Now? - .tell()

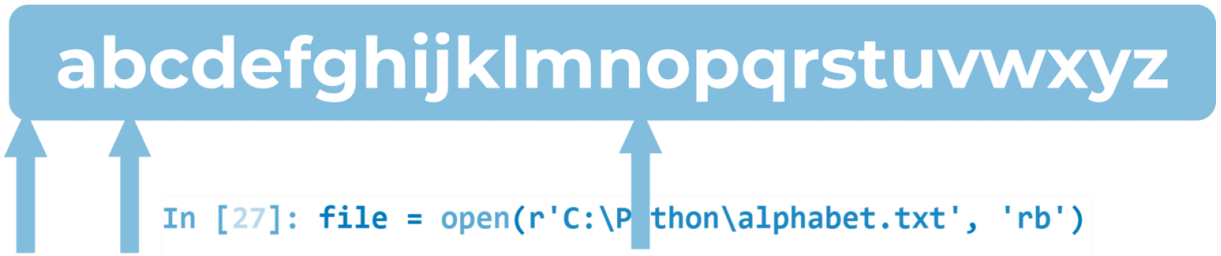
- We can use the .tell() method to ask where the pointer is currently placed:

```
In [23]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [24]: file.read(3)
Out[24]: b'abc'

In [25]: file.read(5)
Out[25]: b'defgh'

In [26]: file.tell()
Out[26]: 8
```

Where is the pointer now? 8 bytes in!

Changing the Pointer Position

- I can change the pointer's position by using the method .seek() and giving the position to change to.




```
In [27]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [28]: file.read(5)
Out[28]: b'abcde'

In [29]: file.seek(2)
Out[29]: 2

In [30]: file.read(3)
Out[30]: b'cde'

In [31]: file.seek(14)
Out[31]: 14

In [32]: file.read()
Out[32]: b'opqrstuvwxyz'
```

Reading the File Twice

- So, in order for us to reread the file (use the .read() method twice), all we need to do is to reset the pointer to the beginning of the file!

```
In [7]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [8]: file.read()
Out[8]: b'abcdefghijklmnopqrstuvwxyz'

In [9]: file.seek(0)
Out[9]: 0

In [10]: file.read()
Out[10]: b'abcdefghijklmnopqrstuvwxyz'

In [11]: file.close()
```

Summary

- Up until now, opening a file took a few lines, and you had to make sure you remember to close the file.

```
def read_file_content(file_path):
    file = open(file_path, 'rb')
    data = file.read().decode()
    file.close()
    return data
```

The with Keyword - Reading

- Using with, we can handle files easier!

```
def read_file_content(file_path):
    with open(file_path, 'rb') as f:
        return f.read().decode()
```

- The with creates the open() object and saves it in the variable f.
- It also makes sure to execute .close() for you, even if an error happens!

The with Keyword iteration by lines

- In Python, a file object iterates over the lines of a file by default. Each line is treated as a separate string element in the iteration. This is commonly used for reading text files line by line.
- For example, consider the following code:

```
# Open a file in text mode for reading
with open("file.txt", "r") as file:
# Iterate over the lines in the file
for line in file:
        # Process the line
        print(line.strip())
```

The with Keyword - Writing

```
def write_file_content(file_path, string_to_write):
    with open(file_path, 'wb') as f:
        f.write(string_to_write.encode())
```

Modes Without ‘b’

- Apart from modes ‘rb’ and ‘wb’, there are the modes ‘r’ and ‘w’.
- When working with data stream, like from the `requests` module, it is better to not use modes ‘r’ and ‘w’ !
- Explanation:
  - We want to maintain the *Unicode encoding* of the text
  - Whenever we read or write plain text files, there are certain bytes that make the writing/reading stop when the pointer reaches them
  - We will usually write and read files that are not plain text, which means that these bytes should be ignored.

Asymmetric Encryption: The RSA algorithm

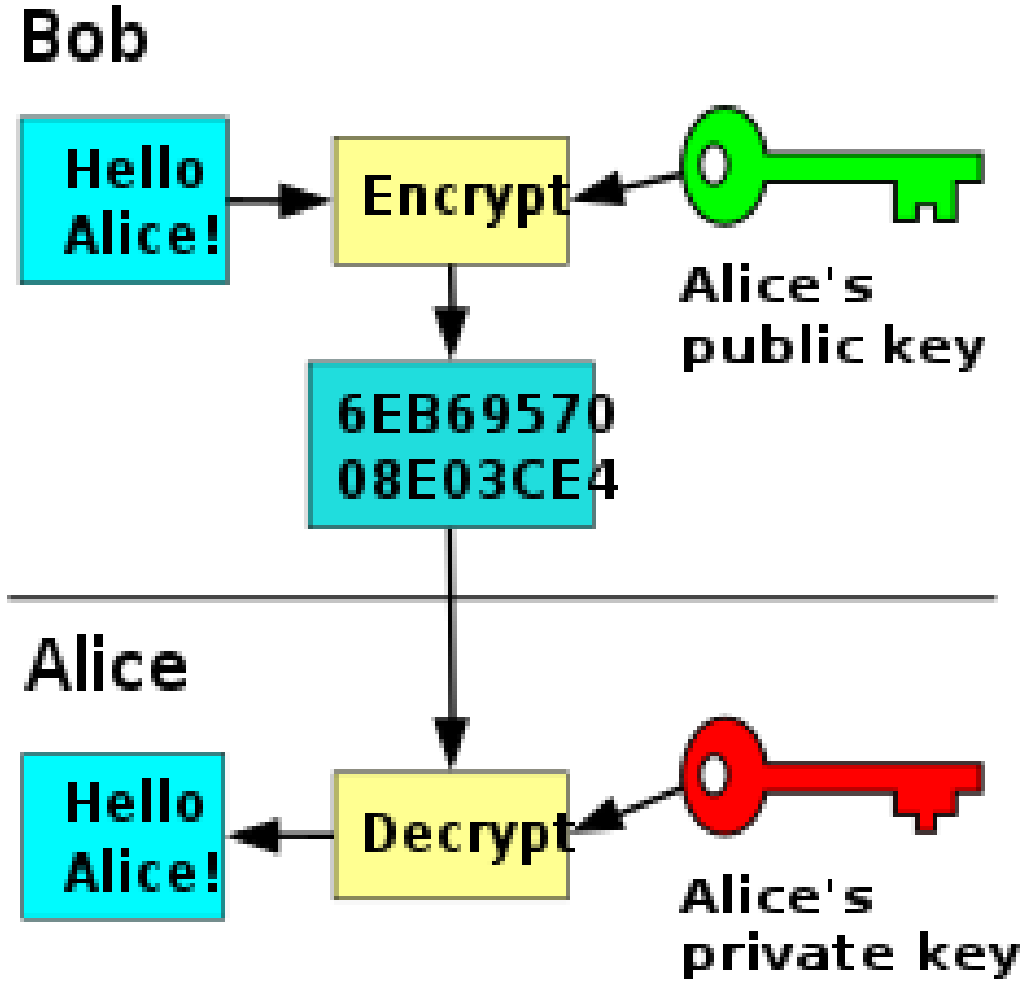
RSA Encryption

- **RSA encryption** is a widely used public-key cryptographic algorithm that allows secure communication over the internet.
- Named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman, RSA encryption is based on the mathematical concepts of prime numbers and modular arithmetic.
- It is widely used for secure data transmission, digital signatures, and key exchange in various applications, including online banking, secure messaging, and e-commerce.

How RSA Encryption Works

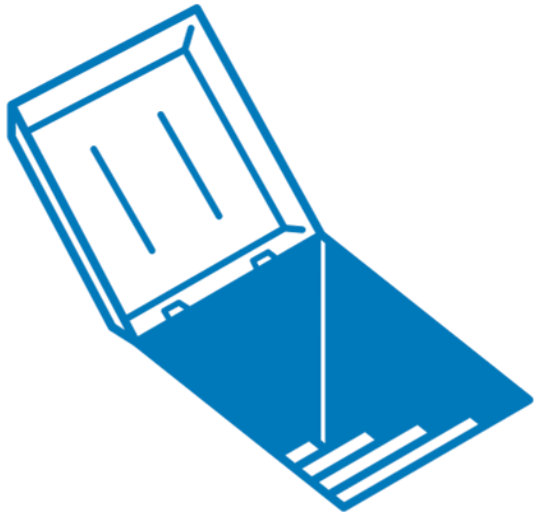
- RSA encryption uses a pair of keys: a **public key** and a **private key**.
- The public key is used for encryption, and it is freely shared with others.
- The private key is kept secret and is used for decryption.
- The process of RSA encryption involves the following steps:
  - Key Generation: The user generates a pair of keys - a public key and a private key.
  - Encryption: The sender uses the recipient's public key to encrypt the plaintext message.
  - Decryption: The recipient uses their private key to decrypt the ciphertext message and obtain the original plaintext.

How RSA Encryption Works Visual



One-way Functions

- One-way functions are a fundamental concept in modern cryptography.
- One-way functions are mathematical functions that are **easy to compute in one direction**, but computationally **difficult to reverse**.
- Also known as **trapdoor** functions, as they allow for efficient computation in one direction, but are practically infeasible to reverse without a special "trapdoor" information.



The RSA Algorithm

- Selecting two large prime numbers, p and q.
- Calculating **n = p * q**, which is used as the modulus for both public and private keys.
- Calculating Euler's totient function, **φ(n) = (p - 1)(q - 1).**
- Choosing an integer **e** such that **1< e < φ(n)** and **gcd(e, φ(n)) = 1**, which becomes the public key exponent.
- Calculating **d**, the modular multiplicative inverse of **e modulo φ(n)**, which becomes the private key exponent.
- Public key is **(n, e)** and private key is **(n, d).**

Key Generation

Select p, q	p and q both prime
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n), e) = 1;\ 1 < e < \phi(n)$
Calculate d	$d \equiv e^{-1} \bmod \phi(n)$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

Encryption

Plaintext	$M < n$
Ciphertext	$C = M^e \pmod{n}$

Decryption

Ciphertext	C
Plaintext	$M = C^d \pmod{n}$

Security of RSA Encryption

- RSA encryption is considered secure because it is based on the difficulty of factoring large composite numbers into their prime factors.
- The security of RSA encryption relies on the length of the keys used. Longer keys provide higher security but also require more processing power for encryption and decryption.
- However, with the advent of quantum computers, which can efficiently factor large numbers, RSA encryption may become vulnerable to attacks in the future.
- It is important to use appropriate key lengths and regularly update keys to maintain the security of RSA encryption.

Cryptography libraries in Python

Python Library For Encryption

Library	Main Characteristics
rsa	- Provides functions for generating RSA key pairs, encrypting and decrypting messages using RSA algorithm. Widely used for secure communication and digital signatures.
cryptography	- Provides a wide range of cryptographic recipes and primitives in a high-level, easy-to-use interface. Actively maintained and widely used for cryptography in Python applications.
pycrypto	- Deprecated library for cryptography in Python, no longer actively maintained or recommended for new projects. Not recommended for use in new projects due to lack of active maintenance and security updates.

Example…

```
>>> import rsa
>>> (public_key, private_key) = rsa.newkeys(2048)
>>> type(public_key)
<class 'rsa.key.PublicKey'>
>>> type(private_key)
<class 'rsa.key.PrivateKey'>
>>> data = b"Hello, red team!"
>>> encrypted_data = rsa.encrypt(data, public_key)
>>> print(encrypted_data)
…snip…
>>> decrypted_data = rsa.decrypt(encrypted_data,
private_key)
>>> print(decrypted_data)
b'Hello, red team!'
```

Security Concerns

Security

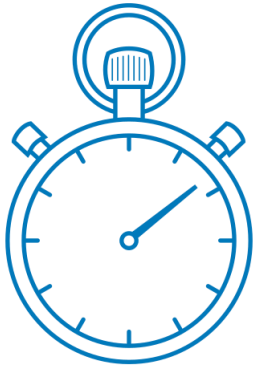
Because of how Python internally stores numbers, it is very hard (if not impossible) to make a pure-Python program secure against timing attacks. This library is no exception, so use it with care. See https://securitypitfalls.wordpress.com/2018/08/03/constant-time-compare-in-python/ for more info.

https://pypi.org/project/rsa/

- *What are timing attacks?*

Timing Attacks in Python

- Timing attacks exploit the variation in execution time of code to infer sensitive information.
- In Python, timing attacks can occur when comparing strings or performing other operations that take different amounts of time depending on the input data.
- Timing attacks can be used to leak information such as passwords, encryption keys, or other confidential data.
- **Example…**



Learning Objectives

- You will be able **to program** simple cipher algorithms, like the Caesar cipher and the transposition cipher
- You will be **able to read** and **to write** files to the file system with Python
- You will be able **to break** substitution ciphers by frequency analysis
- You will be able **to use** the RSA cryptography library in Python
- You will be able **to understand** timing attacks against python login inputs