EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Chapter 1. Python for Network Automation

Contents

- Should network engineers have the programming ability?
- Data types
- Conditional statements
- Understanding Containment
- Loops

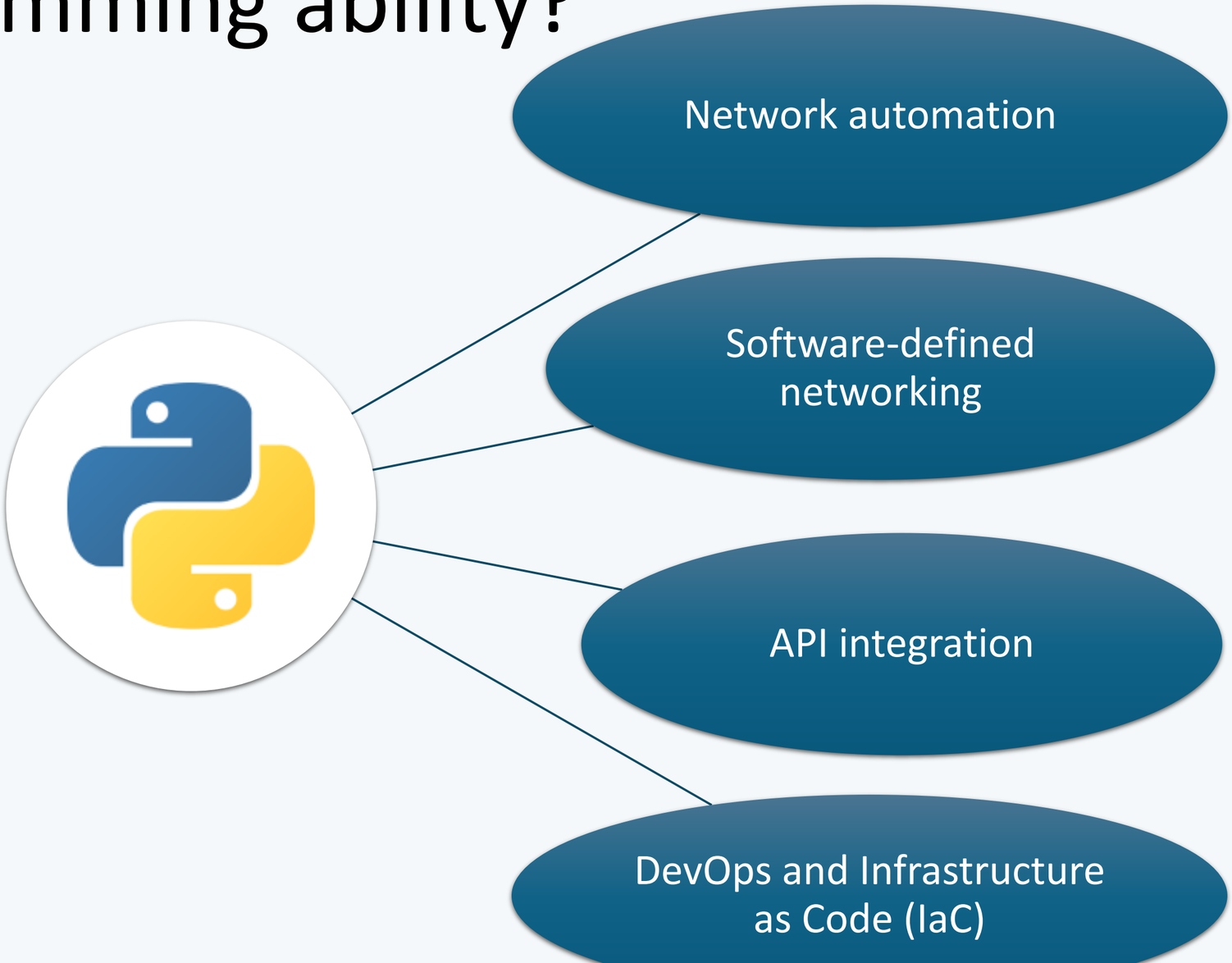
Contents

- Functions
- Files
- Modules
- Pip

Should network engineers have the programming ability?

- Learning the basics of any programming language is valuable.
- Every network engineer should know how to read and write a basic script

Should network engineers have the programming ability?



Python

- In the **network automation area**, **Python** is a powerful and versatile programming language.
- Used to automate, configure, and manage network devices and infrastructure.

Python – key roles in network automation

- Automating Repetitive Tasks:
  - device configuration, firmware updates, and routine checks
- Interacting with Network Devices:
  - Connects to devices via protocols like SSH, Telnet, and REST APIs.
- Network Configuration Management:
  - configure switches, routers, and firewalls programmatically.

Python – key roles in network automation

- Monitoring and Troubleshooting:
  - Collects real-time data from devices using SNMP or APIs.
  - Scripts analyze logs, metrics, and alerts for faster issue resolution.
- Integration with Automation Frameworks:
  - Orchestrates network automation
  - Facilitates Infrastructure as Code (IaC) workflows.
- API Interaction:
  - Retrieves and modifies network settings through device or controller APIs.

Interactive Python

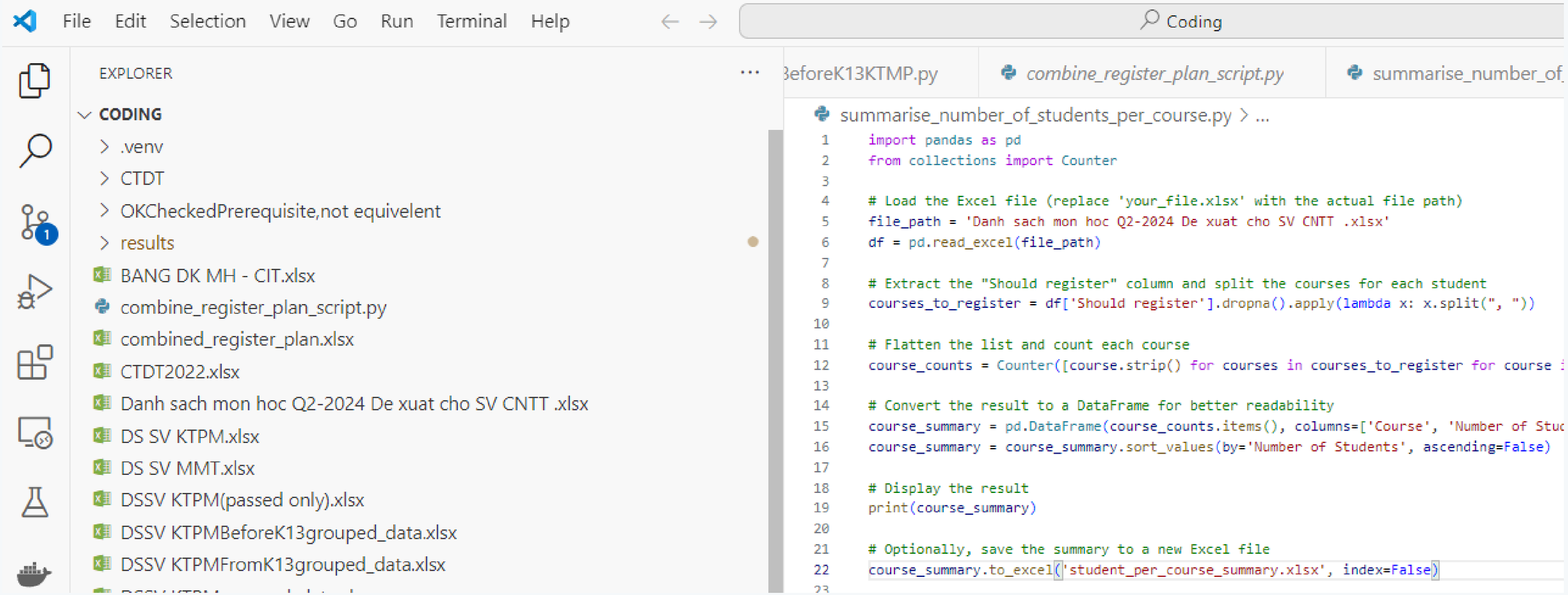
To start interactive mode, type “python” or “python3” in cmd

```
C:\Users\nguye>python
Python 3.10.6 (tags/v3.10.6:9c7b4bd, Aug  1 2022, 21:53:49) [MSC v.1932 64 bit
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello')
hello
>>>
```

Then

```
>>> banner = "\n\n   WELCOME TO ROUTER_1   \n\n"
>>>
>>> print(banner)


   WELCOME TO ROUTER_1



>>>
```

Can Python code for project project creation



Data Type	Usage	Example	Popular Methods/ Functions	Example Usage
int	Represents whole numbers	x = 42	+, -, *, // (floor division), % (modulus), pow()	x = 10 // 3 (floor division gives 3)
float	Represents decimal (floating-point) numbers	pi = 3.14	+, -, *, /, round(), abs()	round(3.14159, 2) gives 3.14
complex	Represents complex numbers with real and imaginary parts	z = 2 + 3j	.real, .imag, abs()	z.real returns 2.0
str	Represents sequences of characters (text)	name = "Python"	.upper(), .lower(), .replace(), .find(), .split(), .strip()	"hello".upper() gives "HELLO"
list	Ordered, mutable collection of items	nums = [1, 2, 3]	.append(), .remove(), .pop(), .sort(), .reverse()	nums.append(4) adds 4 to the list
tuple	Ordered, immutable collection of items	coords = (10, 20)	.count(), .index()	coords.index(20) gives 1

Data Type	Usage	Example	Popular Methods/ Functions	Example Usage
dict	Unordered, mutable collection of key-value pairs	person = {"name": "John"}	.keys(), .values(), .get(), .items(), .update()	person.get("name") gives "John"
set	Unordered, mutable collection of unique items	unique = {1, 2, 3}	.add(), .remove(), .union(), .intersection()	unique.add(4) adds 4 to the set
frozenset	Immutable version of a set	frozen = frozenset([1, 2, 3])	.union(), .intersection(), .issubset(), .issuperset()	frozen.union({4}) gives frozenset({1, 2, 3, 4})
bool	Represents Boolean values (True or False)	flag = True	and, or, not	not True gives False
datetime	Represents date and time objects	from datetime import datetime	.now(), .strftime(), .fromtimestamp()	datetime.now() gives the current date and time

Python data types: key notes

- **Dynamic Typing**:
  - Python is dynamically typed, meaning you don’t need to explicitly declare data types.
  - x = 10 # x is automatically an int
  - x = "hello" # Now x is a string
- **Type Checking:**
  - Use type() to check the data type of a variable.
  - type(42) # <class 'int’>
- **Type Conversion**:
  - Convert between types using int(), float(), str(), etc.

Conditional statements

- If:

```
>>> if hostname == 'NYC':
...     print('This hostname is NYC')
...     print(len(hostname))
...     print('The End.')
```

Conditional statements

- Elif:

```
>>> hostname = 'NJ'
>>>
>>> if hostname == 'NYC':
...     print('This hostname is NYC')
... elif hostname == 'NJ':
...     print('This hostname is NJ')
...
```

Conditional statements

- If-elif-else:

```
>>> hostname = 'DEN_CO'
>>>
>>> if hostname == 'NYC':
...     print('This hostname is NYC')
... elif hostname == 'NJ':
...     print('This hostname is NJ')
... else:

...     print('UNKNOWN HOSTNAME')
...
UNKNOWN HOSTNAME
```

Containment

- The ability to check whether some object contains a specific element or object

```
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> 'arista' in vendors
True
>>>
```

Loops: while

- The general premise behind a while loop is that some set of code is executed while some condition is true.

```
>>> counter = 1
>>>
>>> while counter < 5:
...         print(counter)
...         counter += 1
...
```

Loops: for

- Very useful for looping, or iterating, over a set of objects, like those found in a list, string, or dictionary.
- For other languages, for loop requires an index and an increment
- Python: NOT

Loops: for

```
>>> vendors
['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for vendor in vendors:
...     print('VENDOR: ' + vendor)
...
VENDOR:  arista
VENDOR:  juniper
VENDOR:  big_switch
VENDOR:  cisco
>>>
```

Loops: for

```
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco', 'oreilly']
>>>
>>> approved_vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for vendor in vendors:
...     if vendor not in approved_vendors:
...         print('NETWORK VENDOR NOT APPROVED: ' + vendor)
...
NETWORK VENDOR NOT APPROVED:  oreilly
>>>
```

```
>>> COMMANDS = {
...     'description': 'description {}',
...     'speed': 'speed {}',
...     'duplex': 'duplex {}',
... }
>>>
>>> print(COMMANDS)
{'duplex': 'duplex {}', 'speed': 'speed {}', 'description': 'description {}'}
>>>
```

```
>>> CONFIG_PARAMS = {
...     'description': 'auto description by Python',
...     'speed': '10000',
...     'duplex': 'auto'
... }
>>>
```

```
>>> commands_list = []
>>>
>>> for feature, value in CONFIG_PARAMS.items():
...     command = COMMANDS.get(feature).format(value)
...     commands_list.append(command)
...
>>> commands_list.insert(0, 'interface Eth1/1')
>>>
```

Loops: for

```
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for index, each in enumerate(vendors):
...     print(index + ' ' + each)
...
0 arista
1 juniper
2 big_switch
3 cisco
>>>
```

Counting loops

```
for i in range(10):
    print(i)
```

Python function

- Some code needs to rewrite many times if using script only

Python function

```
# process_january.py

total_sales = 0
tax_rate = 0.07  # 7% tax

# Hardcoded filename for January
with open('sales_january.csv', 'r') as f:
    # Skip header
    next(f)
    for line in f:
        # Assumes format is always product,price,quantity
        parts = line.strip().split(',')
        price = float(parts[1])
        quantity = int(parts[2])
        total_sales += price * quantity

total_tax = total_sales * tax_rate
grand_total = total_sales + total_tax

print(f"January Report:")
print(f"  Subtotal: ${total_sales:.2f}")
print(f"  Tax (7%): ${total_tax:.2f}")
print(f"  Grand Total: ${grand_total:.2f}")
```

The Problem: Now, your boss asks for the February report. What do you do?
You copy process_january.py to process_february.py and change **one line**: with open('sales_february.csv', 'r') as f:.

Python function

A better approach is to abstract the logic into a reusable function. This avoids duplicating code.

```python
def generate_sales_report(filename, tax_rate):
    """
    Calculates sales totals from a given CSV file.
    """
    total_sales = 0
    with open(filename, 'r') as f:
        next(f) # Skip header
        for line in f:
            parts = line.strip().split(',')
            price = float(parts[1])
            quantity = int(parts[2])
            total_sales += price * quantity

    total_tax = total_sales * tax_rate
    grand_total = total_sales + total_tax

    print(f"Report for {filename}:")
    print(f"  Subtotal: ${total_sales:.2f}")
    print(f"  Tax ({tax_rate:.0%}): ${total_tax:.2f}")
    print(f"  Grand Total: ${grand_total:.2f}")
    print("-" * 20)

# Now, you just call the function for each file
generate_sales_report('sales_january.csv', 0.07)
generate_sales_report('sales_february.csv', 0.07)
generate_sales_report('sales_march.csv', 0.07)
```

Python function helps to promote

- **Reusability**:
  - Functions allow you to write code once and use it multiple times.
- **Modularity**:
  - Functions break a program into smaller, manageable pieces.
- **Abstraction**:
  - Functions hide complex details, allowing you to focus on the bigger picture.
- **Maintainability**:
  - Easier to debug and maintain modular code.

Python function

```
>>> def print_vendor(net_vendor):
...     print(net_vendor)
...
>>>
>>> vendors = ['arista', 'juniper', 'big_switch', 'cisco']
>>>
>>> for vendor in vendors:
...     print_vendor(vendor)
```

Python function

```
>>> def get_commands(vlan, name):
...     commands = []
...     commands.append('vlan ' + vlan)
...     commands.append('name ' + name)
...
...     return commands
...
>>>
>>> def push_commands(device, commands):
...     print('Connecting to device: ' + device)
...     for cmd in commands:
...         print('Sending command: ' + cmd)
>>>
```

Python function

```
>>> devices = ['switch1', 'switch2', 'switch3']
>>>
```

```
>>> vlans = [{'id': '10', 'name': 'USERS'}, {'id': '20', 'name': 'VOICE'},
{'id': '30', 'name': 'WLAN'}]
>>>
```

Python function

```
>>> for vlan in vlans:
...     id = vlan.get('id')
...     name = vlan.get('name')
...     print('\n')
...     print('CONFIGURING VLAN:' + id)
...     commands = get_commands(id, name)
...     for device in devices:
...         push_commands(device, commands)
...         print('\n')
...
>>>
```

File IO: Reading Text Files

- To open a text file and read its contents into your program
- The open() function & the with statement: easiest way to open a file and **automatically handles closing** the file for you
- Syntax: with open('filename.txt', mode='r') as file_variable:
  - 'r' mode: Read-only (default mode). The program will raise an error if the file doesn't exist.

File IO: Reading Text Files

- Once the file is open, you can read it in several ways
- Iterate line by line (most common & memory-efficient):

```
# Reads the file one line at a time
with open('data.txt', 'r') as f:
    for line in f:
        print(line.strip()) # .strip() removes leading/trailing whitespace,
```

File IO: Reading Text Files

- Once the file is open, you can read it in several ways
- Read the entire file into a string:

```
# Good for small files
with open('data.txt', 'r') as f:
    content = f.read()
    print(content)
```

File IO: Reading Text Files

- Once the file is open, you can read it in several ways
- Read all lines into a list:

```
# Each list element is one line from the file
with open('data.txt', 'r') as f:
    lines = f.readlines()
    # lines is now ['line 1\n', 'line 2\n', ...]
```

File IO: Writing & Appending to Text Files

- To create new files or add content to existing ones:
  - **'w' (Write Mode**): Creates a new file. Warning: If the file already exists, it will be completely overwritten!
  - **'a' (Append Mode**): Adds content to the end of an existing file. If the file doesn't exist, it will be created

File IO: Writing & Appending to Text Files

- Writing a string (.write()): writes a single string to the file, must manually add newline characters (\n) if want line breaks.

```
# Overwrites 'output.txt' or creates it
with open('output.txt', 'w') as f:
    f.write("Hello, World!\n")
    f.write("This is the second line.")
```

File IO: Writing & Appending to Text Files

- Appending to a file: Use 'a' mode to add text without deleting existing content.

```
# Adds a new line to 'output.txt'
with open('output.txt', 'a') as f:
    f.write("\nThis line was appended.")
```

Python modules

- Any Python source file can be used as a module and any functions and classes you define in that source file can be reused by other Python scripts.
- To load the code, the file referencing the module needs to use the **`import`** keyword

Python modules

- When the file is imported:
    1. The file creates a new namespace for the objects defined in the source file.
    2. The caller executes all the code contained in the module.
    3. The file creates a name within the caller that refers to the module being imported. The name matches the name of the module.

Python modules

- Syntax for importing a module:

```
"import" module ["as" identifier] ("," module
["as" identifier])*
```

Python modules

```
# Import math and time module
import math, time

# Get the factorial of a number using the math module
result = math.factorial(4)
print("Factorial of 4:",result)

# Get the current time using the time module
localtime = time.asctime( time.localtime(time.time()) )
print ("Local current time :", localtime)
```

Python modules

- **The from...import Statement**:
- `"from" relative_module "import" identifier ["as" identifier](","identifier ["as" identifier])*`
- **Example**:

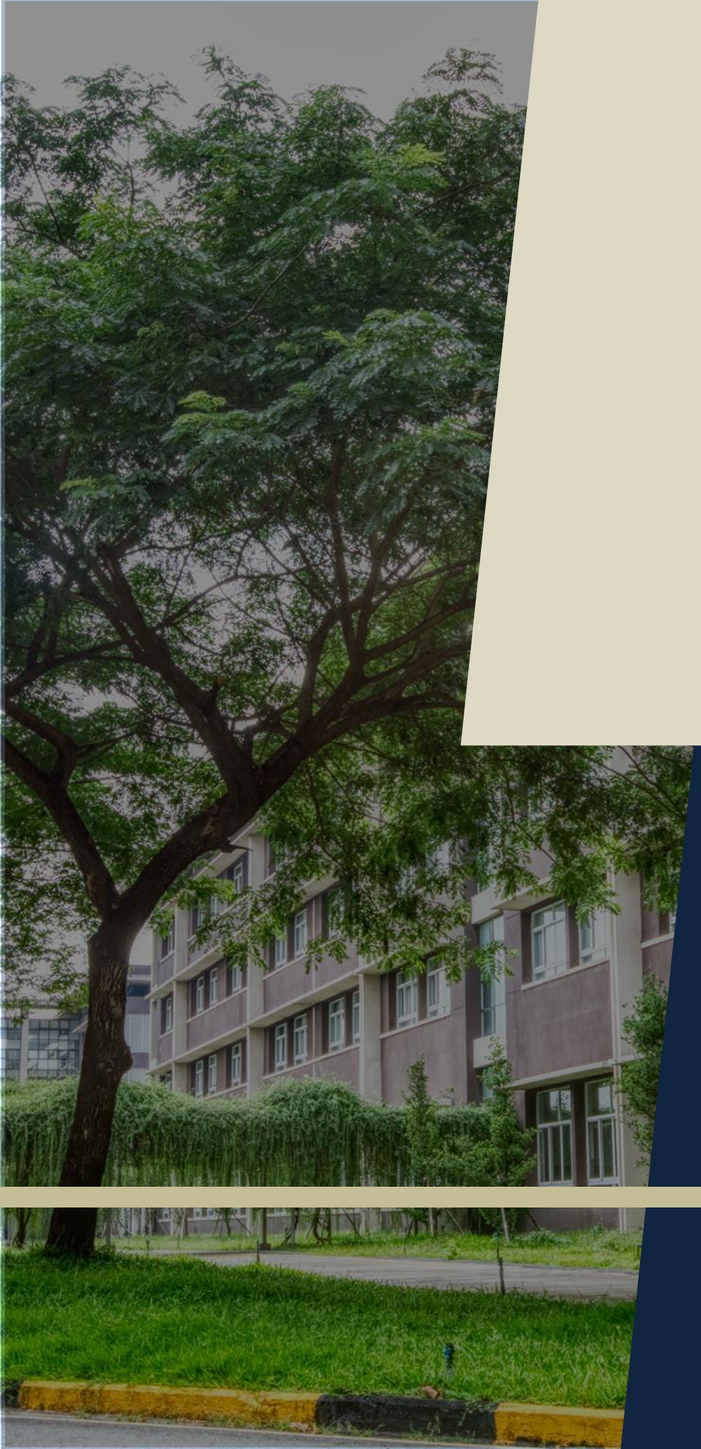
```
# Only import the factorial function
from math import factorial


# Get the factorial of a number
result = factorial(4)
print("Factorial of 4:",result)
```

Python pip

- Used to install and manage packages (libraries or modules) that are not part of the Python standard library.
- `pip install <package_name>`
- **Example:** `pip install numpy`
- `pip uninstall <package_name>`
- **Example:** `pip uninstall numpy`

Python pip

- Installing packages from a requirements file:
  - `pip install -r requirements.txt`
- Generating a requirements file:
  - pip freeze > requirements.txt
- Listing installed packages:
  - pip list



EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

TRÂN TRỌNG CÁM ƠN