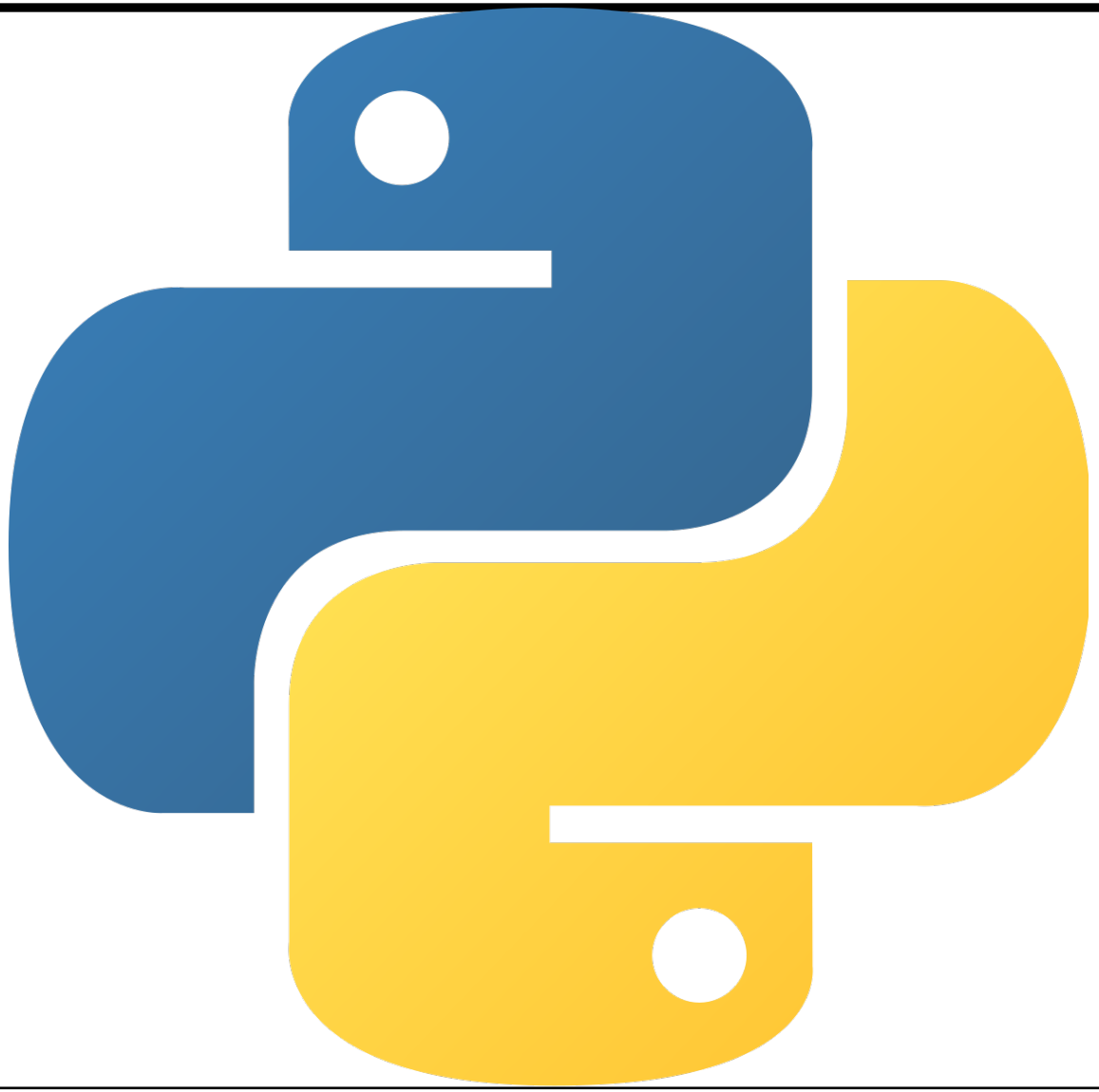
DAGENS FRÅGA

- Om alla hade en varningsetikett, vad skulle det stå på din?

PYTHON PROGRAMMERING



Föreläsning 7

DAGENS AGENDA

- Module och libraries
- Packages
- File handling
- Read from files
- Write to files

FÖRRA FÖRELÄSNING

- Introduktion kursen: studieplanering och kursmaterial
- Bekantade med datorens terminal: Powershell
- Installerade Python med Anaconda (minicona)
- Aktiverade Virtual Environments
- Installerade Visual Studio Codes (VSC)
- Aktiverade Python i VSC

MENTIMETER

8855 1273

MODULE / LIBRARIES

```python
# Saved in a file called mymodule.py
def greeting(name):
    print("Hello ", name)
```

- A module in python sometimes referred to as a library is a collection of definitions that is held in a separate file. That can be included in a script
- A modules allows you to logically organize you python code. Grouping related code into a module makes it easier to understand and allows it to be reused in different programs
- A simple module is a file containing python code ending with a .py extension
- Python has a huge set of standard modules make sure that if you need something specific to search for it first as someone might already have solved it for you

IMPORT MODULES

```
# import file mymodule.py as a module
import mymodule
import math as maths # part of standard module
from os import name # import function name from module os

# call the function greeting in the module mymodule
mymodule.greeting("World")
print(maths.floor(2.4)) # rounds value down prints 2
print(name) # prints OS name
```

- To include a module you can use the *import* keyword which will add the name of the module to the global scope
- When importing local files that are in directories you use the from keyword to move into a directory and import for the file. For example **sub/mod.py** would be imported with *from* **sub** *import* **mob**
- When importing a module you can change the name of it by using the *as* keyword
- To import something specific from a module you can use the *from .. import* keywords

PACKAGES

- To further organize your modules you can use packages. A package can import several modules at once.
- In practice a package typically corresponds to a file directory containing python files and other directories. To create a python package yourself, you create a directory and a file names __init__.py inside it
- In __init__.py you can define imports for that package this can be used to include subdirectories or modules automatically when import a package. When importing files in __init__.py you should use *from* **.** *import* ***module***

```
# foo/__init__.py
print("__Init__ running")
from . import bar
# foo/bar.py
print("Imported bar")
def hello():
    print("Hello bar")
#foo/baz.py
print("Imported baz")
# packages.py
import foo
#Prints
# __Init__ running
# Imported bar
foo.bar.hello() #Prints hello bar
```

FILE HANDLING

- When working with files you can use the built-in *open()* function. The *open* function returns a file object that contains methods and properties to perform various operations on files.
- The *open("filename", "mode")* function takes two parameters. The name of the file that should be opened(can also be a path to a file). The mode dictates what can be done to the file and what should happen if the file does not exist.
- From the OS side a file handle is created for the file and assigned to the running process. This blocks other processes from doing certain operations on the file such as editing it. Because this creates an external handle it must be closed so that the OS knows to release the file handle. This is done using the *close()* function.
- When the python process is closed the file handle will automatically be released by the garbage collector but it's better to release file handles as soon as you are done with them.

```
f = open("test.txt", "w+")
# Open or creates the file text.txt with mode w+
f.write("Test line")
# writes Test line to the file
f.close()
# closes the file handle so that other processes can use it


# safe way to use open close
try:
    f = open("try_test.txt", "w")
    f.write("Test Line")
finally:
    # will always be closed
    f.close()
```

FILE MODES

Mode	Description
r	Default mode. Opens file for reading. Places file pointer at beginning
w	Opens file for writing. If file does not exist, it creates a new file. If file exists it truncates the file(emptys it) Places file pointer at beginning
a	Open file in append mode. If file does not exist, it creates a new file. Places file pointer at end of the file
b	This opens in binary mode.
+	This will open a file for reading and writing (updating)

- Python have many file modes which makes it easier and clearer what should be done.
- A file pointer determines where in a file operations take place. Certain mode only moves the file pointer to specific places for example w and r will place the file pointer at the beginning of the file and a will place in at the end of the file.
- Modes can be combined to increase what can be done with the file. r, w, a are main modes that can't be combined with each other. b and + can be added to all the main modes to increase their behavior. For example rb+ would open a file in binary mode for both reading and writing with the file pointer placed at the beginning of the file

READING FROM FILES

- Python have three ways to read from files: *read()*, *readline()* and *readlines()*.
- *read(count)* will read as many character as count if left empty will read the entire file. r*eadline()* reads a single line if the end of file was reached it returns ""(empty string). r*eadlines()* reads the entire file and splits them into lines
- You can also iterate directly over the file object using *for line in f:* I use this instead of *readline()* as it's less verbose

```
# read entire file
f = open("file.txt", "r")
print(f.read())
# prints Hello
# World
f.close()

# read file line by line
f = open("file.txt")
for line in f:
    if line == "World":
        print("Found", line)
        # prints Found World
f.close()
```

WRITING TO FILES

- Python have two way to write data to a file *write() and writelines().*
- *write()* writes a string to the file.
- Note that it does not add a newline if you want a newline add the newline character(\n).
- w*ritelines()* writes a sequence to the files. No line endings are appended to each sequence element.
- If you want to append to the end of a file use the append mode.

```
f = open("file.txt", "w")
f.write("Hello\n")
f.write("World")
f.close()
# If it was not closed next open would cause exception
```

THE *WITH* STATEMENT

- When using certain python objects that create or use external things(such as files ,network connection, database connection etc.) you can use the *with* statement.
- The *with* statement will make it easier to handle files as it will remove the need to explictly call close and use a try/finally block.
- When leaving the *with* statement the closing function of the object will be called no matter how you leave the with statement.
- It also makes it clearer where a file is open as it creates an indent where the file can be used.
- Always use *with* statements when it's possible to do so
- You can make your own objects *with* compliant

```
with open("with_test.txt", "w+") as f:
    f.write("Test line")
# file handle is closed when moving out of block
print(f.read()) # Raise exception
# ValueError: I/O operation on closed file
```