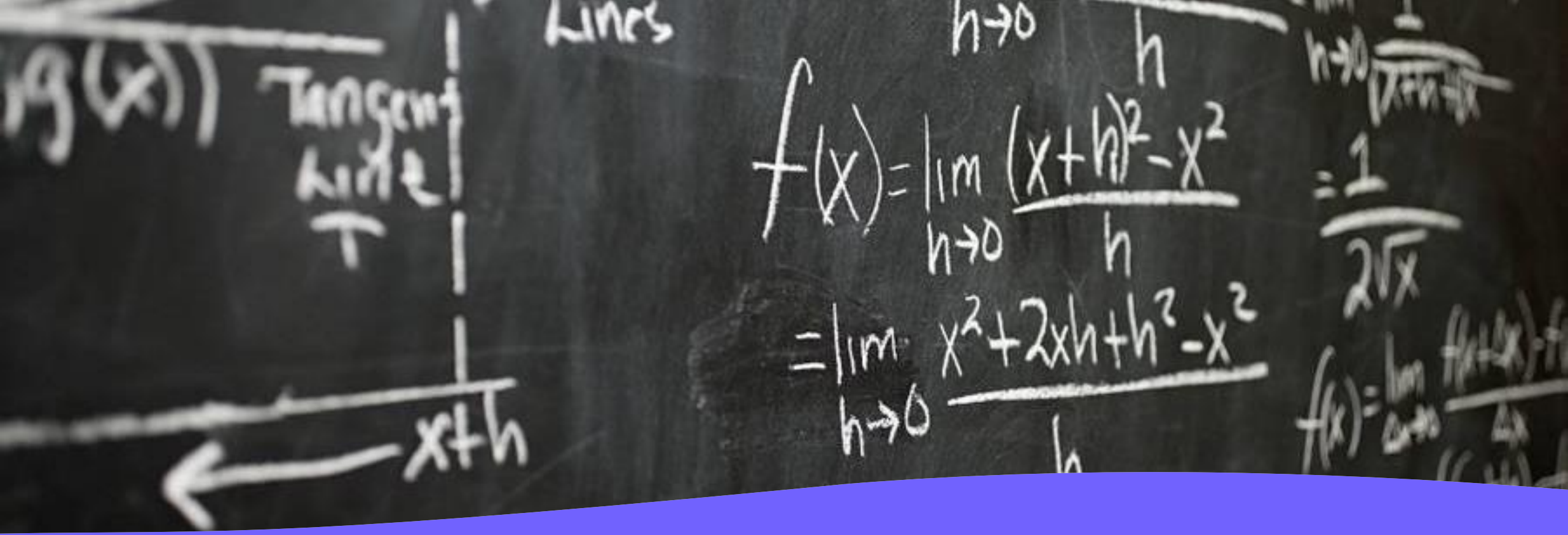



# COMP1112: Week 11

PROF NAME HERE

# Agenda

Writing & Using Custom Modules

Servers and Scheduling

Web Scraping Bot

Lab: Create A Web Scraping Bot

Wrap Up

# Threads & Processes

# Threads & Processes

So far most of our programs have been sequential, with a little bit of variation when we run async tasks. But some of the power in a computer comes from it being able to do many complex things at once. A computer would not be very useful if you had to close a stackoverflow page to open VS code! How would we ever get anything done?

Instead of making a few method calls at the same time with async, sometimes we want to run entire programs at the same time with their own dedicated resources. Threads allow us to parcel out programs into instructions that can be executed at the same time, with their own dedicated memory. Let's quickly brush up on how they work.

# Threads 1

Most modern computers represent processes as “threads”. A web browser will load an image on one thread and use another to format the web page. The Python Virtual Machine runs on several threads, opening the IDLE editor in terminal happens on one thread, your main Python application runs on another, and the Python garbage collector runs on its own thread too.

Python treats threads like any other object, it can hold data, be stored in data structures, and be passes as method arguments.

Some code defined in a thread can also be executed as a process. To execute the code, a thread must implement the `run` method.

Threads can be in different states over their lifetime. Once a thread is created it remains **newborn**, and will stay inactive until its `start` method is run. Once `start` is run the thread changes to **ready** and in placed in the **ready queue**.

Before the thread can be executed it must wait in the queue for the CPU to be free to process it. Once a thread has access to a CPU it starts running instructions in its `run` method.

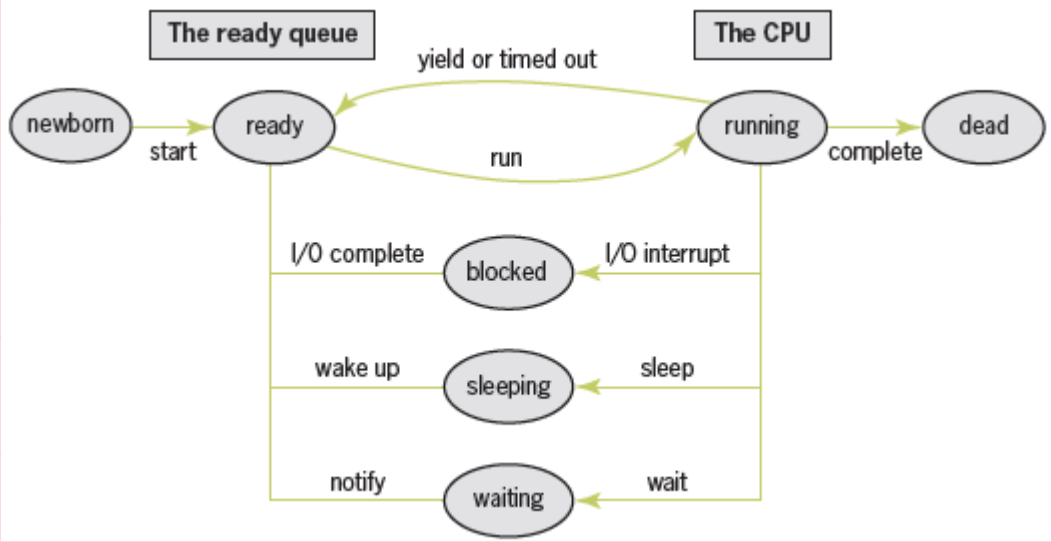
# Threads 2

Threads can lose access to the CPU in 4 different ways:

1. **Time-Out**: most computers running python automatically time out a thread every few milliseconds, placing it at the back of the ready queue and running the next thread
2. **Sleep**: A thread can be put to sleep for a given number of milliseconds. When it wakes up it goes to the back of the ready queue.
3. **Block**: A thread can wait for some event like user input, When a blocked thread is notified that an event has occurred it goes to the back of the ready queue
4. **Wait**: A thread can voluntarily relinquish the CPU until some condition is true. When the condition becomes true the thread is moved to the back of the ready queue.

# Threads 3

Here is a diagram describing how threads are executed from page 355 of the text:




Let's look at threadingtest.py to see how Pythons threading module lets us create our own threads.

# Thread Methods

A table of Thread methods from page 356 of the text:

| Thread Method | What It Does |
|---|---|
| `__init__(name = None)` | Initializes the thread's name. |
| `getName()` | Returns the thread's name. |
| `setName(newName)` | Sets the thread's name to `newName`. |
| `run()` | Executes when the thread acquires the CPU. |
| `start()` | Makes the new thread ready. Raises an exception if run more than once. |
| `isAlive()` | Returns `True` if the thread is alive or `False` otherwise. |

Please note this is a little out of date, so you will get a warning if using getName/setName. Instead we can access the attribute directly with the following syntax:

```
print(aThread.name)          # Getting and printing name with modern syntax
aThread.name = "New Name"    # Setting a new name with modern syntax
```

# Sleeping Threads

For this example we will create a simple program that lets a user start several threads. Once started, each thread will sleep for a random number of seconds, once it wakes it will print a message then terminate.

Note that when a thread goes to sleep, the next thread in the ready queue will have an opportunity to use the CPU. Because of the randomized sleep times, we expect to see varying results from run to run.

Open **sleepythreads.py** and let's debug together (also on page 357-258 of the text).

# Writing & Using Custom Modules

# Custom Modules 1

Throughout the course we have used a lot of powerful Python modules at no cost. This is due to the strong community of developers behind Python, in fact the language was developed with **extensibility** in mind.

Even if you aren't writing code for others, modules are still useful to help break your program out into parts. Like a math problem, programs become easier when you break them out into smaller parts. You may find down the road some of the modules you design for one program are useful in another!

# Custom Modules 2

A great example of something we do all the time is CSV processing. We could write our own CSV processor and it would be much easier to translate between Python arrays and the strings that hold CSV data.

To see what I mean, lets look at `csv_processor.py` first, then `test_csv.py` to see it in action.

Custom modules can do whatever you would like, but sometimes the freedom makes them more difficult to write. When you are writing a module its important to make sure it has a specific purpose (ex: manage CSV files, connect to the internet) and it can accomplish that purpose all on its own.

# Breaking Down a Program

When you write a program, you will usually start with a complex problem like: “I want to collect competitor price data over time”. Just like a word problem in math class, we need to break this down into smaller parts until we have individual modules we can either create or use premade modules for. Let’s break this problem down together:

**Collect:** this will involve making web requests of some sort, as well as web scraping and HTML parsing.

**Competitor price data:** this data will need to be stored/loaded somewhere.

**Over time:** the bot will need to run perpetually collecting data at regular intervals, this will require server and scheduling modules.

# Breaking Down a Program

**Collect:**

- Requests (HTTP requests)
- BeautifulSoup4 (Web scraper)
- Html5lib (Web parser)

**Competitor price data:**

- csv_processor.py (custom module from 2 slides ago)

**Over time:**

- A server module
- A time/scheduling module

# Breaking Down a Program

Once we have all these modules assembled, we would write our main.py which will run the program for us using the imported/copied modules.

Main should take care of any of the specific logic. In our example main will tell the CSV processor which files to load/save, tell the requests module which websites to request, process the data returned with beautiful soup, etc.

# Servers and Scheduling

# Servers & Scheduling

A server is usually defined as a computer that provides some functionality over the web. In this class when we talk about a server, we are talking about the type of program that would run on a server. Specifically, we will learn to design a scripts that are always on and awaiting some sort of request/trigger to begin processing.

Our Discord bot was a server in this regard, it was running in a loop awaiting Async events from the Discord server. Today we will make a server that runs a web scraping bot twice per day. The actual server will handle the orchestration (it decides which program to run based on a provided trigger) and the bot will run on its own thread.

# Flask

We are going to set up a simple server using Flask. Flask allows us to create a lightweight server with just a few lines of code. For now, we will set up a simple server on your local machine. We will use port 8080, the default port for most web servers.

Before we do anything else, install flask with `pip install flask`

Let's look at `demoServ.py` together.

We can run demo serv from the command line with `flask --app demoServ.py run`

This will start our demo server, to use the server open your browser and navigate to: http://127.0.0.1:5000/

# Web Scraping Bot

# Creating a Keep-Alive server

To ensure our bot continues running indefinitely we will create a keep-alive server. This is a server that will stay running on a separate thread to ensure our task continues running. Let's check out `keepAliveServ.py` for the details.

# Creating Our Bot

To run our bot we need to do some scheduling work on the frontend. Specifically, we need to create a schedule and calculate the time until it runs, then run the bot after the specified amount of time. If all the runs are over for the day, we will wait 4 hours and run the check again.

If there are runs left for the day, we wait the specified amount of time, scrape the data we need, then save it to a CSV file using our CSV processor module.

We do some of this scheduling with the datetime module, it has some important features like `datetime.datetime.now()` which gets the current date and time, as well as `datetime.timedelta(hours, minutes)` lets us create a timespan we can use to add, subtract, and compare. We can also convert timedeltas to floats with `timedelta.total_seconds()` which gets the time span as a float in seconds.

We also use the time module which has `time.sleep(seconds)` allowing us to sleep for a specified number of seconds

Let's go over `scrapeBot.py` and see how we can use the keep-alive server we designed to run our bot on a schedule.

# Lab: Create A Web Scraping Bot

In the lab for class 9 we wrote a script to scrape a website of your choosing. Use this script as a starting point and do the following:

- Turn your web scraping script into a module that can be used by a server.
- Write a server which runs the script 4 times a day (1 AM, 9 AM, 1 PM, 9 PM)

If you run into an issue remember to debug and read the error message carefully. When you run into an error your first reaction should always be debugging!

This guide on calculating time differences should help: https://geekflare.com/calculate-time-difference-in-python/

# Quiz 2

Quiz 2 will be available next week, it is worth 5% of your mark. It is meant to prepare you for the final exam so expect them to be similar. This Quiz covers weeks 8 - 11.

# Final Exam

The final exam is worth **30% of your final mark**. It covers **weeks 1 - 12** including *labs, assignments, and readings*.

Expect a large part of the questions to be code writing and comprehension, mixed with other question types (multiple choice, true/false, etc). It will be similar to the mid term.

# Up Next

**This Week's Reading:** -

**Next Week's Topic:** Multithreaded Servers and PDFs

**Due Next Week:** Quiz 2

# Thank you

PROF NAME

Prof.Email@GeorgianCollege.ca