CS131 - Week 8

UCLA Winter 2019
TA: Kimmo Karkkainen

Today

- A brief introduction to Python
- Asyncio
- Project

Python Introduction

Python

- General-purpose, interpreted language
  - Very popular, easy to write code
  - Libraries for nearly every purpose, from machine learning to web servers
- We will use Python 3
  - Python 2 still somewhat common, even though Python 3 was released a decade ago
- Download from: https://www.python.org
  - If you want to use Python for data science, consider downloading Anaconda package instead https://www.anaconda.com/download/

Python resources

- https://www.learnpython.org/
  - Interactive tutorial, fast and easy to get started
- https://docs.python.org/3/tutorial/
  - Official tutorial
- https://docs.python.org/3/
  - Reference material for the language and the official libraries

Hello World

```
print(“Hello World!”)
```

Hello World!

Variables

- Dynamic typing, no special syntax for declaring a new variable

```
x = 123                          # integer
x = 123L                         # long integer
x = 3.14                         # double float
x = "hello"                      # string
x = [0,1,2]                      # list
x = (0,1,2)                      # tuple
x = open(‘hello.py’, ‘r’)        # file
```

Functions

- Declared using *def* keyword:

```
def my_function(name):
        print("Hello, " + name)

my_function("Steve")
```

Functions - Positional vs Keyword Parameters

```
def my_func(a, b=1, c=1):
        print(a + b + c)

my_func(2)  # “4”

my_func(2, 2, 2)  # “6”

my_func(1, c=2)  # “4”
```

Typical program structure

```
def main():
    print(“Hello World!”)

if __name__ == '__main__':
    main()
```

- Notice the lack of parentheses/curly braces, indentation matters
- Also, no semi-colons at the end of the line (line breaks matter)
- Python does not have a main function by default, so we need to call it manually

Python Modules

- Every file will define a module, for example:

mymodule.py:

```
def print_hello(name):
        print("Hello, " + name)
```

main.py:

```
import mymodule

def main():
        mymodule.print_hello("Steve")

if __name__ == '__main__':
    main()
```

$ python main.py
Hello, Steve

__name__ == “mymodule”

Python Modules

- You can also import functions to avoid using *mymodule.print_hello(...)*:

main.py:

```
from mymodule import print_hello

def main():
        print_hello("Steve")

if __name__ == '__main__':
    main()
```

Variable scope

```
x = 5

def my_function():
    print(x)

my_function()
```

Output:

5

```
x = 5

def my_function():
    x = 10
    print(x)

my_function()
print(x)
```

Output:

10
5

Variable scope

```
x = 5

def my_function():
    print(x)
    x = 10

my_function()
print(x)
```

Output:

ERROR

```
x = 5

def my_function():
    global x
    print(x)
    x = 10

my_function()
print(x)
```

Output:

5
10

Classes

```
class Person:
        def __init__(self, name, age):
                self.name = name
                self.age = age

        def print_greeting(self, greeting):
                print(greeting + self.name)

p = Person("John", 36)
p.print_greeting("Hello, ")
```

Output:

Hello, John

Lists

- Lists are dynamic length arrays (compare to Scheme/OCaml/Prolog)
  - Fast random access, easy to add/remove elements (with a slight performance overhead)
  - Uses a bit more memory

```
my_list = [1, 2, 3]

print(my_list[2])
3

print(my_list[1:])
[2, 3]
```

```
for item in my_list:
        print(item)
1
2
3

for idx, item in enumerate(my_list):
        print(idx, item)
0 1
1 2
2 3
```

Lists - map/filter

```
new_list = [x**2 for x in my_list]
print(new_list)
```
[1, 4, 9, 16, 25]

```
new_list2 = [x for x in my_list if x%2==0]
print(new_list2)
```
[2, 4]

OCaml map:
```
List.map (fun x -> x*x) [1; 2; 3; 4; 5];;
```

OCaml filter:
```
List.filter (fun x -> x mod 2 = 0) [1; 2; 3; 4; 5];;
```

Generators

```
def my_func():
        n = 0
        while True:
                n += 1
                yield n

a = my_func()
print(next(a))
print(next(a))
print(next(a))
```

Output:

1
2
3

Generators - map

```
powers = (2**x for x in range(0, 1000000000))

print(next(powers))
print(next(powers))
print(next(powers))
print(next(powers))
```

Output:

1
2
4
8

Dictionary

```
my_dict = { "a": 1, "b": 2, 42: 3 }

print(my_dict["a"])
1

print(my_dict[42])
3

my_dict["something new"] = 4
print(my_dict["something new"])
4
```

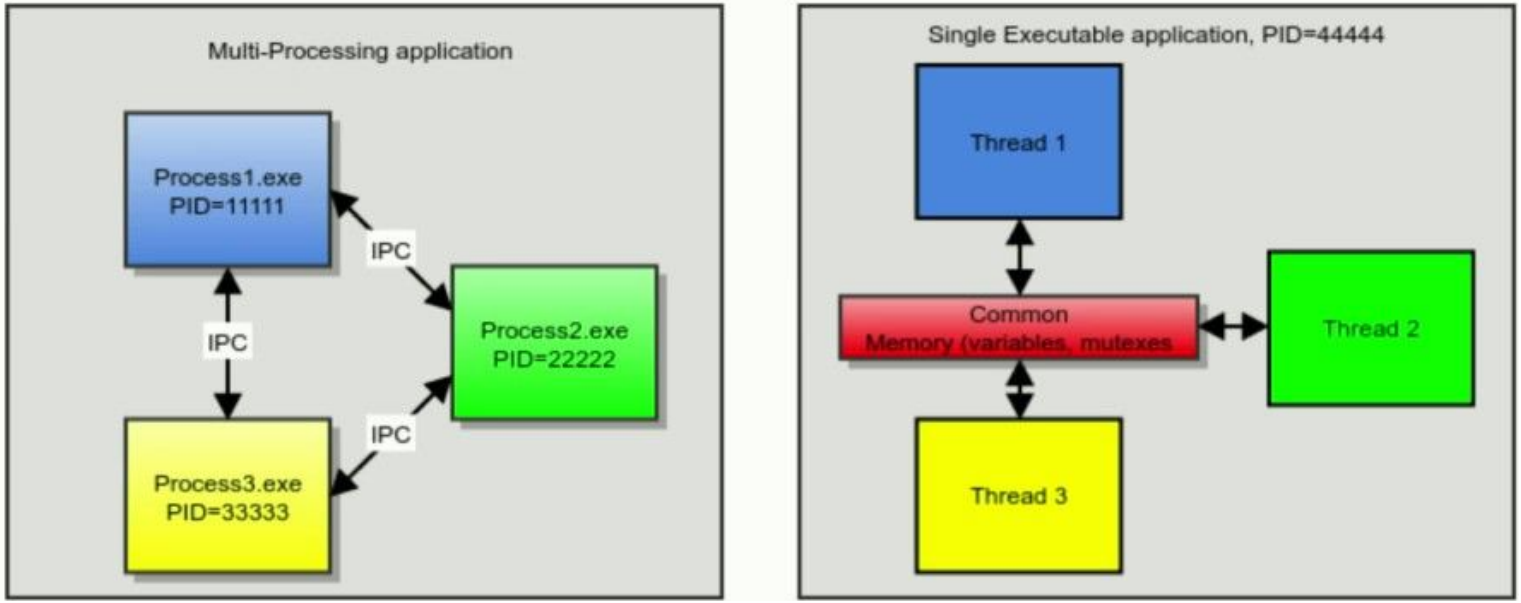
```
my_dict.keys()
dict_keys(['a', 'b', 42])

for k in my_dict.keys():
        print(k, my_dict[k])
a 1
b 2
42 3
```

Asyncio

Multithreading vs Multiprocessing

- What’s the difference?

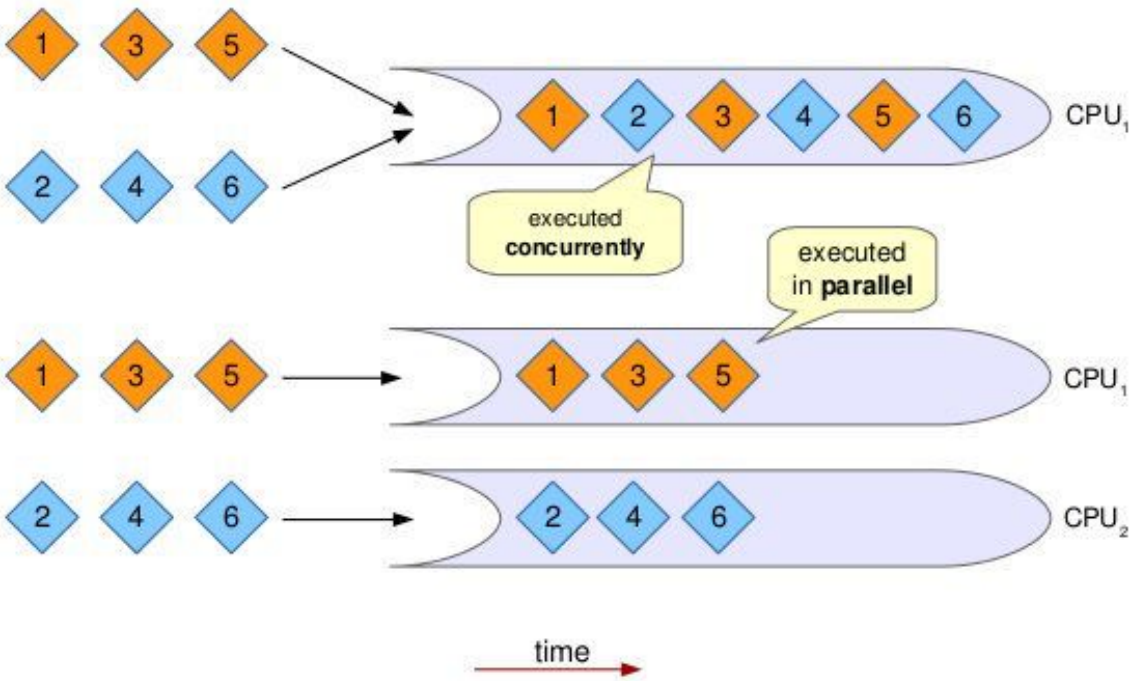
Multithreading vs Multiprocessing



Parallelism vs Concurrency

- What’s the difference?

Parallelism vs Concurrency



Global Interpreter Lock (GIL)

- A lock that can be held by only one thread at a time
  - Owner of that lock is allowed to execute, no other thread can run simultaneously
- Why?
  - Python memory management depends on reference counting -> possible race conditions
    - Compare to other garbage collection approaches - Pros/Cons?
  - Using separate lock for all reference counts could be inefficient and cause deadlocks
  - Python also uses C libraries that are not thread-safe

```
>>> import sys
>>> a = []
>>> b = a
>>> sys.getrefcount(a)
3
```

Consequences of GIL

- Fast single-threaded code
  - Simple memory management compared to other types of garbage collection
- Multithreading does not improve the performance of CPU intensive tasks
  - Might be even slower due to locks!
- When would we benefit from threads in Python?

How to utilize multiple cores with Python?

- Multiprocessing
- Libraries
  - E.g. Many numerical computation and machine learning libraries support parallel processing
    - Implemented in C or other low-level language

Asyncio

- Cooperative multitasking library
  - Tasks can voluntarily take breaks and let other tasks run
  - Compare to *preemptive multitasking*
- Single-threaded approach for concurrent programming (not parallel!)
  - Very similar to multithreading, but not the same
- Introduced in Python 3.4 (2014)
  - Relatively new, so changes often with new versions (we use 3.7, which is the latest)

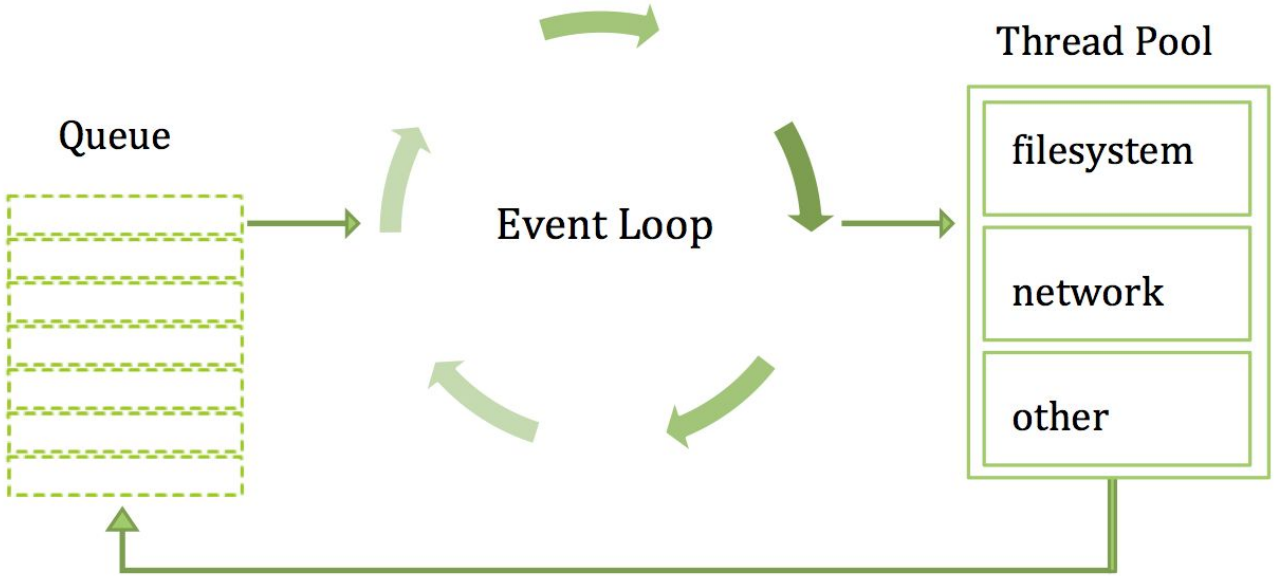
Basic concepts

- *Async* keyword
  - Defines that a function is a *coroutine*
  - A function that can suspend its execution and give control to another coroutine
- *Await* keyword
  - Suspends the execution of the current coroutine until the *awaited* function is finished

```
async def g():
    r = await f()
    return r
```

Event loop

- Event loop runs tasks that are waiting



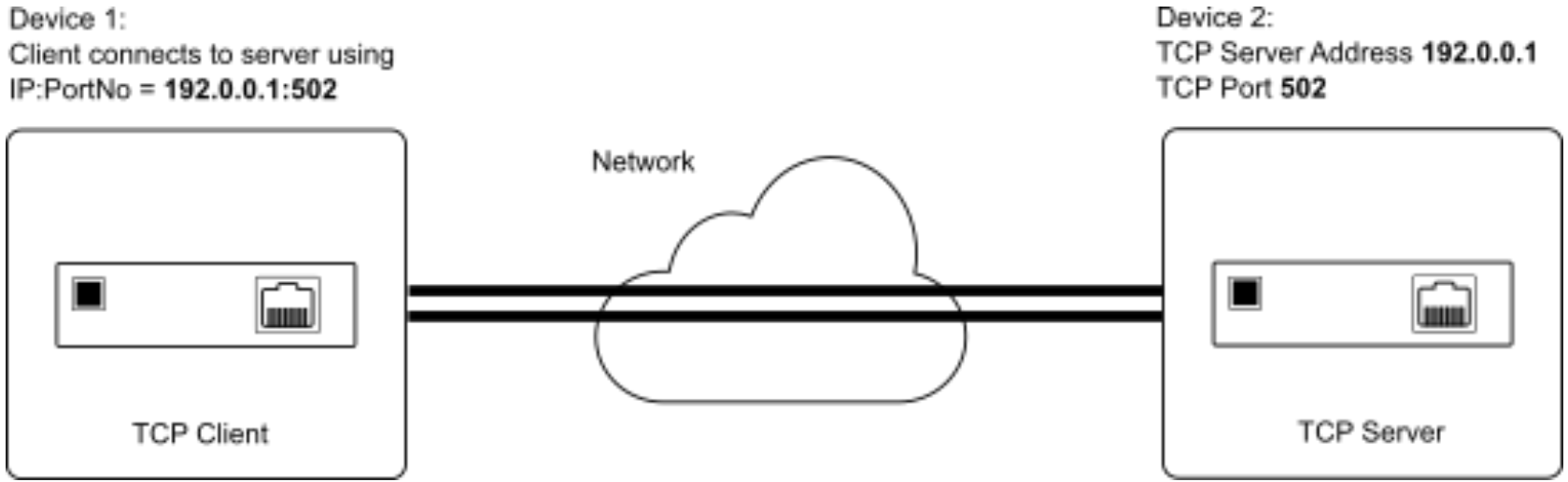
Example

```
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)  # Any IO-intensive task here
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())  # Add to an event loop
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```
$ python3 countasync.py
One
One
One
Two
Two
Two
countasync.py executed in 1.01 seconds.
```

Implementing Servers with Asyncio

- Asyncio can be used to write TCP clients/servers
  - TCP provides reliable connections
    - error detection, ordering, ...



Implementing a Server with Asyncio

```
import asyncio

async def main():
    server = await asyncio.start_server(handle_connection, host='127.0.0.1', port=12345)
    await server.serve_forever()

async def handle_connection(reader, writer):
    data = await reader.readline()
    name = data.decode()
    greeting = "Hello, " + name
    writer.write(greeting.encode())
    await writer.drain()
    writer.close()

if __name__ == '__main__':
    asyncio.run(main())
```

Implementing a Server with Asyncio

```
import asyncio

async def main():
    server = await asyncio.start_server(handle_connection, host='127.0.0.1', port=12345)
    await server.serve_forever()

async def handle_connection(reader, writer):
    data = await reader.readline()
    name = data.decode()
    greeting = "Hello, " + name
    writer.write(greeting.encode())
    await writer.drain()
    writer.close()

if __name__ == '__main__':
    asyncio.run(main())
```

Implementing a Client with Asyncio

```
import asyncio

async def main():
    reader, writer = await asyncio.open_connection('127.0.0.1', 12345)
    writer.write("John\n".encode())

    data = await reader.readline()
    print('Received: {}'.format(data.decode()))

    writer.close()


if __name__ == '__main__':
    asyncio.run(main())
```

Implementing a Client with Asyncio

```
import asyncio

async def main():
    reader, writer = await asyncio.open_connection('127.0.0.1', 12345)
    writer.write("John\n".encode())

    data = await reader.readline()
    print('Received: {}'.format(data.decode()))

    writer.close()


if __name__ == '__main__':
    asyncio.run(main())
```

```
sh-3.2$ python client.py
Received: 'Hello, John\n'
```

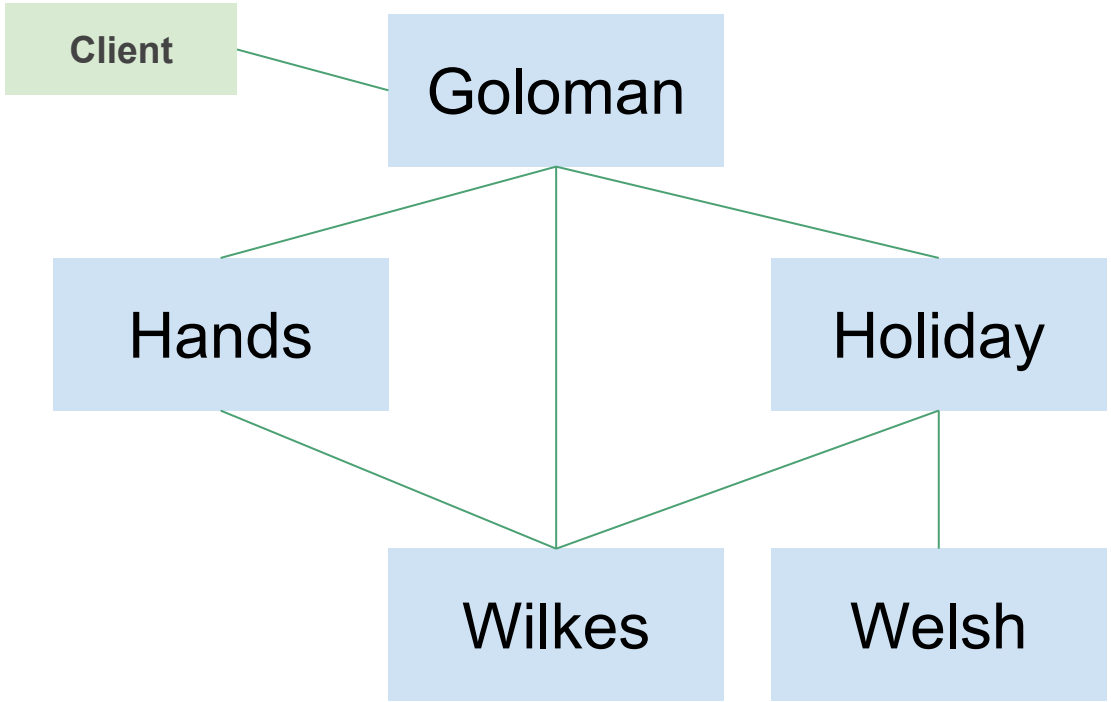
Asyncio Resources

- Async IO in Python: A Complete Walkthrough
- How the heck does async/await work in Python 3.5?
- Asyncio Documentation

Project

Project (DL 3/8)

Task: Build a server herd that can synchronize data and communicate with client applications



Client-Server Communication - IAMAT

- Clients can send their current location to any server using TCP protocol:

IAMAT kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997

Name of the command

Client ID

Coordinates

Timestamp (POSIX)

- Server responds:

AT Goloman +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997

Name of the response

Server name

Timestamp diff

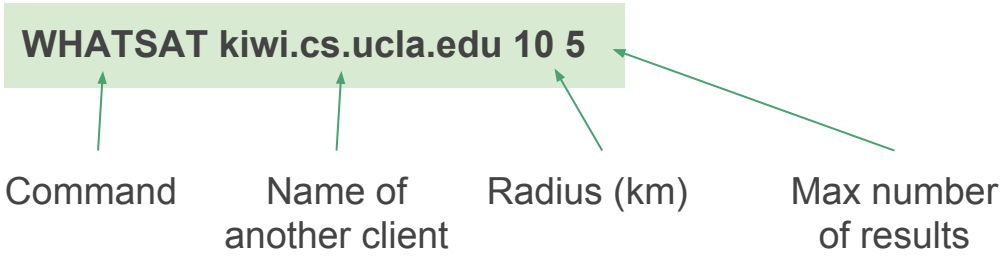
Copy of the client data

Server-Server Communication

- After a server receives a location, it must inform the other servers about the updated location
- Implement a flooding algorithm so that every server receives the message, even if it is not directly connected to the original server
  - Challenge: Must prevent infinite loops
  - You can decide what type of messages servers use to communicate
- If a server goes down, all the other servers should still function normally
  - No need to propagate old messages when the server is restarted

Client-Server Communication - WHATSAT

- Clients can ask what is near one of the clients:



Client-Server Communication - WHATSAT

- Server uses Google Places API to find the nearby locations
  - Google Places API gives results in JSON format, return it to the client in the same format, just remove duplicate newlines (see project instructions for details)
  - Append the usual “AT ...” line before the Google’s response

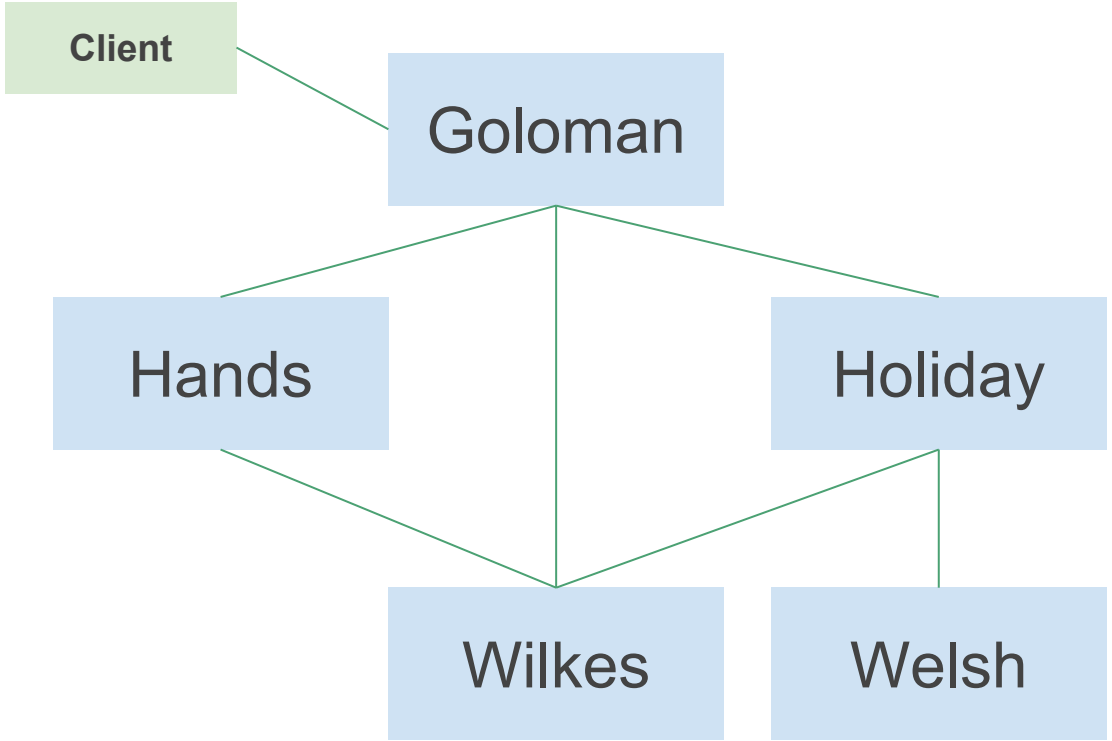
Client-Server Communication - WHATSAT

- Server responds:

```
AT Goloman +0.263873386 kiwi.cs.ucla.edu +34.068930-118.445127 1520023934.918963997
{
   "html_attributions" : [],
   "next_page_token" : "CvQ...L2E",
   "results" : [
      {
         "geometry" : {
            "location" : {
               "lat" : 34.068921,
               "lng" : -118.445181
            }
         },
         "icon" : "http://maps.gstatic.com/mapfiles/place_api/icons/university-71.png",
         "id" : "4d56f16ad3d8976d49143fa4fdfffbc0a7ce8e39",
         "name" : "University of California, Los Angeles",
         "photos" : [
            {
               "height" : 1200,
               "html_attributions" : [ "From a Google User" ],
```

Recap

- Client can send:
  - IAMAT
  - WHATSAT
- IAMAT:
  - Server saves the location and propagates it
- WHATSAT:
  - Server calls Google Places API to check what is near the given user, sends results to caller



Google Places API

- https://cloud.google.com/maps-platform/places/
- Gives you information on what is around a given location
  - Also can be used to find an address for given coordinates, get details on specific locations
- You need to create a developer account to access the API
  - Free trial is enough for this project
  - Do not share your key with anyone! (Including Github..)

Google Places API

- Example request:

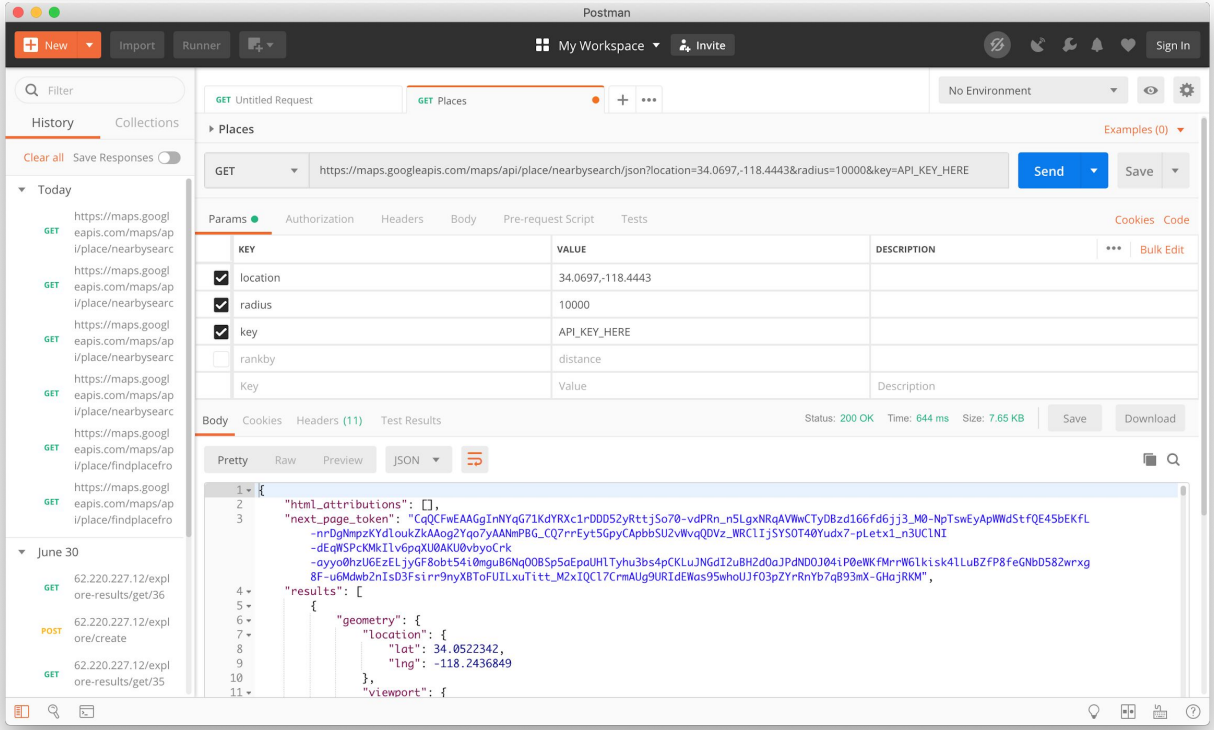
```
https://maps.googleapis.com/maps/api/place/nearbysearch/json?location=-33.8670522,151.1957362&radius=1500&type=restaurant&keyword=cruise&key=YOUR_API_KEY
```

- Searches places near coordinates -33.8670522, 151.1957362
- Limits radius to 1500 meters
- Limits searched places to restaurants
- Searches for keyword “cruise” from any information related to that place

See documentation for description of all the search parameters

Testing Google Places API requests

- Postman is a free tool for testing HTTP requests



How to Make HTTP Requests in Python?

- Use aiohttp library
  - This is only for making requests to Google Places API, do not use it for server functionality!

```
async with aiohttp.ClientSession() as session:
        params = [('param-name1', 'some value'), ('param-name2', '100')]

        async with session.get('https://ucla.edu', params=params) as resp:
                print(await resp.text())
```

- Note: you can re-use the same *session* for all the requests

Report

- Max 5 pages
- Discuss pros/cons of *asyncio*
  - Is it suitable for this kind of an application?
  - What problems did you run into?
  - Any problems regarding type checking, memory management, multithreading?
    - Compare to Java
  - How does *asyncio* compare to Node.js?

Project

- We'll let you know what ports to use on SEASnet
  - Every student will have different ports
- Make sure the requests/responses look exactly the same as instructed, as we'll use automated tests to grade the submissions

Questions?