Asyncio lab 6

https://github.com/hanattaw/class6-asyncio.git

1-1-hello-world-wrong.py > ...

```python
# When do coroutines start running?
import asyncio
import time

async def print_after(message, delay):
    """Print a message after the specified delay (in seconds)"""
    await asyncio.sleep(delay)
    print(f"{time.ctime()} - {message}")

async def main():
    # Start coroutine twice (hopefully they start!)
    first_awaitable = print_after("world!", 2)
    second_awaitable = print_after("Hello", 1)
    # Wait for coroutines to finish
    await first_awaitable
    await second_awaitable

asyncio.run(main())
```

1-2-hello-world-time.py > ...

```python
import asyncio
import time

async def example(message):
    print(f"{time.ctime()} - start of :", message)
    await asyncio.sleep(1)
    print(f"{time.ctime()} - end of :", message)

async def main():
    # Start coroutine twice (hopefully they start!)
    first_awaitable = example("First call")
    second_awaitable = example("Second call")
    # Wait for coroutines to finish
    await first_awaitable
    await second_awaitable

asyncio.run(main())
```

1-3-hello-world-task.py > ...

```python
import asyncio
import time

async def print_after(message, delay):
    """Print a message after the specified delay (in seconds)"""
    await asyncio.sleep(delay)
    print(f"{time.ctime()} - {message}")

async def main():
    # Start coroutine twice (hopefully they start!)
    first_awaitable = asyncio.create_task(print_after("world!", 2))
    second_awaitable = asyncio.create_task(print_after("Hello", 1))
    # Wait for coroutines to finish
    await first_awaitable
    await second_awaitable

asyncio.run(main())
```

1-4-hello-world-gather.py > main

```python
import asyncio
import time

async def print_after(message, delay):
    """Print a message after the specified delay (in seconds)"""
    await asyncio.sleep(delay)
    print(f"{time.ctime()} - {message}")

async def main():
    # Use asyncio.gather to run two coroutines concurrently:
    await asyncio.gather(
        print_after("world!", 2),
        print_after("Hello", 1)
    )

asyncio.run(main())
```

2-1-aiohttp-aiofile.py > ...

```python
# pip install aiofiles==0.7.0
# pip install aiohttp==3.7.4.post0

import sys
import asyncio
import time

import aiohttp
import aiofiles


async def write_genre(file_name):
    """
    Uses genrenator from binaryjazz.us to write a random genre to the
    name of the given file
    """

    async with aiohttp.ClientSession() as session:
        async with session.get("https://binaryjazz.us/wp-json/genrenator/v1/genre/") as response:
            genre = await response.json()

    async with aiofiles.open(file_name, "w") as new_file:
        print(f"{time.ctime()} - Writing '{genre}' to '{file_name}'...")
        await new_file.write(genre)


async def main():
    tasks = []

    print(f"{time.ctime()} - Starting...")
    start = time.time()

    for i in range(5):
        tasks.append(write_genre(f"./asyncout/new_file{i}.txt"))

    await asyncio.gather(*tasks)

    end = time.time()
    print(f"Time to complete asyncio read/writes: {round(end - start, 2)} seconds")


if __name__ == "__main__":
    # On Windows, this finishes successfully, but throws 'RuntimeError: Event loop is closed'
    # The following lines fix this
    # Source: https://github.com/encode/httpx/issues/914#issuecomment-622586610
    if sys.version_info[0] == 3 and sys.version_info[1] >= 8 and sys.platform.startswith('win'):
        asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())

    asyncio.run(main())

```

3-1-database-answer.py > asyncpg_async_get_monitors_many_calls

```python
import asyncio
import asyncpg
import time

# Postgres database details:
host = "localhost"
db_name = "northwind"
port = "55432"
username = "postgres"
password = "postgres"
schema = "public"

PINK = '\033[38;5;205m'
TEAL = '\033[38;5;31m'
GREEN = '\033[32m'
RESET = '\033[0m'

def print_pink(msg):
    print(f"{PINK} {time.ctime()} - {msg} {RESET}" )

def print_teal(msg):
    print(f"{TEAL} {time.ctime()} - {msg} {RESET}" )

def print_green(msg):
    print(f"{GREEN} {time.ctime()} - {msg} {RESET}" )

def print_in_color(msg, color):
    if color == 'green':
        print_green(msg)
    elif color == 'teal':
        print_teal(msg)
    else:
        print_pink(msg)

async def asyncpg_async_get_monitors_many_calls(color="green", id="ALFKI"):
    print_in_color(f"Openning connection", color=color)
    conn = await asyncpg.connect(
        host=host,
        port=port,
        user=username,
        password=password,
        database=db_name,
        server_settings={'search_path': schema}
        )
    rows = await conn.fetch(f"""SELECT (SUM((1 - order_details.discount) * order_details.unit_price * order_details.quantity))::NUMERIC::MONEY AS totalamount FROM orders JOIN order_details ON (orders.
    order_id=order_details.order_id) JOIN customers ON (customers.customer_id=orders.customer_id) WHERE customers.customer_id = '{id}'""")
    for row in rows:
        print_in_color(dict(row), color=color)
        endquery = time.monotonic()
        print_in_color(f"time on id '{id}' is {endquery - start} seconds" , color=color)

    await conn.close()

async def main():
    await asyncio.gather(
        asyncpg_async_get_monitors_many_calls(color='green', id='ALFKI'),
        asyncpg_async_get_monitors_many_calls(color="pink", id='ANATR'),
        asyncpg_async_get_monitors_many_calls(color="teal", id='BERGS')
    )

if __name__ == "__main__":
    start = time.monotonic()
    asyncio.run(main())
    end = time.monotonic()
    print(f"total time {end - start} seconds")
```

machine > 1-2-washing-async.py > ...

```python
"""
ต้องการซักผ้า 2 ตะกร้า แบบ asynchronous io
กระบวนการซักผ้า 1 ตะกร้า คือ
1. หยอดเหรียญเพื่อซักผ้า
2. นำผ้าเข้าเครื่องซักผ้า
3. ซักผ้าเสร็จ (ใช้เวลา 5 วินาที)

เนื่องจากมีเครื่องซักผ้าที่สามารถพร้อมใช้งานได้ 2 เครื่องพร้อมกัน

เปลี่ยนการทำงานเป็นแบบ asynchronous io
"""

import time

import asyncio

async def wash(basket):
    print(f'Washing Machine ({basket}): Put the coin')
    print(f'Washing Machine ({basket}): Start washing...')
    await asyncio.sleep(5)
    print(f'Washing Machine ({basket}): Finished washing')
    return f'{basket} is completed'

async def main():
    coro = wash('Basket A')
    print(coro)
    print(type(coro))
    task = asyncio.create_task(coro)
    print(task)
    print(type(task))
    result = await task
    print(result)

if __name__ == '__main__':
    t1 = time.time()
    asyncio.run(main())
    t2 = time.time() - t1
    print(f'Executed in {t2:0.2f} seconds.')
```


```python
"""
ต้องการซักผ้า 2 ตะกร้า แบบ asynchronous io
กระบวนการซักผ้า 1 ตะกร้า คือ
1. หยอดเหรียญเพื่อซักผ้า
2. นำผ้าเข้าเครื่องซักผ้า
3. ซักผ้าเสร็จ (ใช้เวลา 5 วินาที)

เนื่องจากมีเครื่องซักผ้าที่สามารถพร้อมใช้งานได้ 2 เครื่องพร้อมกัน

เปลี่ยนการทำงานเป็นแบบ asynchronous io
"""
```

```
"""
การรัน Task หรือ Coroutine พร้อมกันหลายตัว
เราสามารถใช้ method ในการรัน task หรือ coroutine พร้อมกันหลายงานได้ โดยผมจะขอยกตัวอย่างมา 3 ตัว คือ
1. asyncio.gather()
2. asyncio.wait()
3. asyncio.as_completed()


ซึ่งความแตกต่างระหว่าง 2 ตัวนี้ คือ
- asyncio.gather() จะรันเรียงตามลำดับก่อนหลัง ส่วน asyncio.wait() และ asyncio.as_completed() นั้นไม่เรียง
- asyncio.gather() ไม่สามารถกำหนด timeout ได้ แต่ asyncio.wait() กำหนดได้ผ่านพารามิเตอร์ timeout
- asyncio.wait() สามารถกำหนดเงื่อนไขในการรอได้โดยใช้พารามิเตอร์ return_when
- asyncio.gather() ไม่สามารถรับค่าเป็น list ของ coroutine/task ได้โดยตรงแบบ asyncio.wait() ให้ใช้ * หน้า list เพื่อกระจายเป็น argument แทน
- asyncio.as_completed() จะใช้กับ for Loop ซึ่งไม่เหมือนตัวอื่น และเป็น for Loop ที่ทุกรอบเริ่มทำงานพร้อมกัน

จงเปรียบเทียบผลของการสร้าง Task ระหว่าง gather(), wait() และ as_completed()

"""
```

```
"""
การรัน Task หรือ Coroutine พร้อมกันหลายตัว
เราสามารถใช้ method ในการรัน task หรือ coroutine พร้อมกันหลายงานได้ โดยผมจะขอยกตัวอย่างมา 3 ตัว คือ
1. asyncio.gather()
2. asyncio.wait()
3. asyncio.as_completed()


ซึ่งความแตกต่างระหว่าง 2 ตัวนี้ คือ
- asyncio.gather() จะรันเรียงตามลำดับก่อนหลัง ส่วน asyncio.wait() และ asyncio.as_completed() นั้นไม่เรียง
- asyncio.gather() ไม่สามารถกำหนด timeout ได้ แต่ asyncio.wait() กำหนดได้ผ่านพารามิเตอร์ timeout
- asyncio.wait() สามารถกำหนดเงื่อนไขในการรอได้โดยใช้พารามิเตอร์ return_when
- asyncio.gather() ไม่สามารถรับค่าเป็น list ของ coroutine/task ได้โดยตรงแบบ asyncio.wait() ให้ใช้ * หน้า list เพื่อกระจายเป็น argument แทน
- asyncio.as_completed() จะใช้กับ for Loop ซึ่งไม่เหมือนตัวอื่น และเป็น for Loop ที่ทุกรอบเริ่มทำงานพร้อมกัน

จงเปรียบเทียบผลของการสร้าง Task ระหว่าง gather(), wait() และ as_completed()

"""
```

machine > 1-3-1-microwave-async-answer.py > ...

```
import asyncio
import time

async def cook(food, t):
    print(f'{time.ctime()} - Microwave ({food}): Cooking {t} seconds...')
    await asyncio.sleep(t)
    print(f'{time.ctime()} - Microwave ({food}): Finished cooking')
    return f'{food} is completed'

async def main():
    coros = [cook('Rice', 5), cook('Noodle', 3), cook('Curry', 1)]
    result = await asyncio.gather(*coros)
    print(f"{time.ctime()} - {result}")

if __name__ == '__main__':
    t1 = time.time()
    asyncio.run(main())
    t2 = time.time() - t1
    print(f'Executed in {t2:0.2f} seconds.')
```

machine > 1-3-2-microwave-async-answer.py > ...

```
import asyncio
import time

async def cook(food, t):
    print(f'{time.ctime()} - Microwave ({food}): Cooking {t} seconds...')
    await asyncio.sleep(t)
    print(f'{time.ctime()} - Microwave ({food}): Finished cooking')
    return f'{food} is completed'

async def main():
    coros = [cook('Rice', 5), cook('Noodle', 3), cook('Curry', 1)]
    results = await asyncio.wait(coros, return_when='FIRST_COMPLETED')
    print(f'Completed task: {len(results[0])}')
    for completed_task in results[0]:
        print(f' - {completed_task.result()}')
    print(f'Uncompleted task: {len(results[1])}')


if __name__ == '__main__':
    t1 = time.time()
    asyncio.run(main())
    t2 = time.time() - t1
    print(f'Executed in {t2:0.2f} seconds.')
```

machine > 1-3-3-microwave-async-answer.py > ...

```
import asyncio
import time

async def cook(food, t):
    print(f'{time.ctime()} - Microwave ({food}): Cooking {t} seconds...')
    await asyncio.sleep(t)
    print(f'{time.ctime()} - Microwave ({food}): Finished cooking')
    return f'{food} is completed'

async def main():
    coros = [cook('Rice', 3), cook('Noodle', 3), cook('Curry', 3)]
    for coro in asyncio.as_completed(coros):
        result = await coro
        print(result)

if __name__ == '__main__':
    t1 = time.time()
    asyncio.run(main())
    t2 = time.time() - t1
    print(f'Executed in {t2:0.2f} seconds.')
```

Asynchronous programming of making breakfast.

Demonstrates asynchronous programming in Python using an example of making breakfast. Suppose we have the following instructions to make a breakfast.

- Pour coffee.
- Heat a pan and fry two eggs.
- Toast 2 slices of bread.
- Add butter to toast.

There are two ways you can execute these instructions

1. in the first method, you do each these steps sequentially that is you pour a cup of coffee, then heat the pan and fry the eggs. You wait for the eggs to be ready and then you toast the bread and finally add butter to toast.

Now, if you have cooking experience you will execute some of the steps in this instruction concurrently.

2. So you start by pouring a cup of coffee, then you heat a pan and while the pan is heating you put a slice of bread in the toaster. Then you crack two eggs in the pan. So now you have the eggs and toast cooking at the same time. When the toast is ready you take that out and put another bread in the toaster. You then apply butter on the first toasted bread and move on to take the fried eggs. When the second slice of bread is toasted you apply butter on it and breakfast is ready

! So basically you switch your attention between toasting bread and frying eggs thereby reducing the overall time required to prepare the breakfast.

Compared to the first method, it takes less time to cook your breakfast this way. In the second method, multiple tasks are carried out at the same time and you switch between tasks whenever your current task doesn't need your attention. This is the concept of asynchronous programming.

Start by creating empty classes for each of the breakfast item.

```
class Coffee:
   pass
class Egg:
   pass
class Toast:
   pass
```

Next, we define the tasks. The first task is to pour coffee. This is not an asycnhronous task, in other words we cannot do anything else while pouring coffee. So let's define this task as a normal function. All this function does is to print the message Pouring coffee

```
def PourCoffee():
   print("Pouring coffee")
   return Coffee()
```

The second step is to fry two eggs, which can be done concurrently while toasting the bread. Therefore this task can be defined as a coroutine.

```
async def FryEggsAsync(howMany):
    print("Heat pan to fry eggs")
    await asyncio.sleep(3)
    print("Frying",howMany,"eggs")
    await asyncio.sleep(3)
    print("Eggs are ready")
    return Egg()
```

The final two steps in the instruction is to toast each slice of the bread and apply butter on toast. These two steps are related but applying the butter can be done only after the bread is toasted. Therefore these two steps have to be performed sequentially but can be done in parallel with frying eggs.

```
async def ApplyButter():
   print("Spreading butter on toast")
   await asyncio.sleep(1)

async def ToastAsync(slices):
   for slice in range(slices):
     print("Toasting bread", slice + 1)
     await asyncio.sleep(3)
     print("Bread", slice + 1, "toasted")
     await ApplyButter()
     print ("Toast", slice + 1, "ready")
   return Toast()
```

The two async tasks, i.e, FryEggsAsync() and ToastAsync() can be run concurrently using the function **asyncio.gather()**.

```
 await asyncio.gather(FryEggsAsync(2),ToastAsync(2))
```

Putting all these pieces of code together, we have the full program as below.

```
import asyncio
import time

class Coffee:
    pass
class Egg:
    pass
class Toast:
    pass

def PourCoffee():


async def ApplyButter():


async def FryEggsAsync(howMany):


async def ToastAsync(slices):


async def main():


if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{time.ctime()} - Breakfast cooked
in",elapsed,"seconds.")
```

result

```
Tue Aug  1 22:30:43 2023 - Pouring coffee
Tue Aug  1 22:30:43 2023 - Coffee is ready
Tue Aug  1 22:30:43 2023 - Heat pan to fry eggs
Tue Aug  1 22:30:43 2023 - Toasting bread 1
Tue Aug  1 22:30:46 2023 - Pan is ready
Tue Aug  1 22:30:46 2023 - Frying 2 eggs
Tue Aug  1 22:30:46 2023 - Bread 1 toasted
Tue Aug  1 22:30:46 2023 - Spreading butter on toast
Tue Aug  1 22:30:47 2023 - Toast 1 ready
Tue Aug  1 22:30:47 2023 - Toasting bread 2
Tue Aug  1 22:30:49 2023 - Eggs are ready
Tue Aug  1 22:30:50 2023 - Bread 2 toasted
Tue Aug  1 22:30:50 2023 - Spreading butter on toast
Tue Aug  1 22:30:51 2023 - Toast 2 ready
Tue Aug  1 22:30:51 2023 - Breakfast cooked in 8.00863425
seconds.
```