CS 61A Fall 2025

Concurrency

Discussion 9: October 29, 2025

If there are fewer than 3 people in your group, merge your group with another group in the room. If your group has 6 or more students, you're welcome to split into two sub-groups and then sync up at the end. If you want two separate Pensieve documents for the two sub-groups, just have one sub-group add 1000 to their group number.

Switch to Pensieve:

- **Everyone**: Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

Attendance

Your TA will come around during discussion to check you in. You can start on the worksheet before being checked in; you don't need to wait for your TA to get started.

If you didn't attend for a good reason (such as being sick), fill out this form (within 2 weeks of your discussion): attendance form

Getting Started

Intros: Say your name and your favorite food or beverage to enjoy in Berkeley. Optionally go have that food or beverage as a group after section!

Python Coroutines

- An `async def` statement creates a *coroutine function*. Calling it creates a coroutine.
- Within a coroutine function, `await` another coroutine (or other *awaitable* object) to pause the current coroutine and only start it again some time after the awaited coroutine returns. Use `x = await c` to assign `x` to the return value of the coroutine `c`.
- After an `await`, mutable objects passed into a coroutine may have been changed by another coroutine.
- To start some first coroutine `c`, use `asyncio.run(c)`.
- To pause the current coroutine for `k` seconds and let other coroutines run, use `await asyncio.sleep(k)`.
- To run two coroutines `c` and `d` concurrently until both return and name their return values `c_val` and `d_val`:

  ```
  c_val, d_val = await asyncio.gather(c, d)
  ```

`async def my_function`: Writing `async` before `def` indicates that the function is a *coroutine function*. Similar to generator functions, when a coroutine function is called, the function does not begin executing. Instead, calling a coroutine function returns a coroutine object.

`await`: The `await` keyword pauses execution of the current coroutine until an awaitable object has completed. For this course, we'll focus on `await`ing coroutines, which are one type of awaitable object. An `await` expression may

only be used within a coroutine. It suspends execution of the current coroutine until some time after the coroutine object (or other awaitable object) following the `await` keyword has completed.

The `asyncio.sleep` coroutine suspends execution until the passed-in number of seconds have elapsed.

The following code defines a coroutine function `wait_for` that pauses execution until `seconds` have passed:

```
async def wait_for(seconds):
    await asyncio.sleep(seconds)
```

Coroutines must be run within an *event loop*. The event loop manages the execution of coroutines, and executes a single coroutine at a time. When the currently running coroutine yields control by calling `await`, the event loop executes another coroutine, if one is available. To start an event loop, use the `asyncio.run` function:

```
asyncio.run(wait_for(5))
```

To run many coroutines concurrently, use `asyncio.gather`, which runs the given coroutines concurrently and waits until all have completed.

```
async def concurrent_wait_fors():
    await asyncio.gather(wait_for(5), wait_for(5), wait_for(5))
```

`wait_for` uses `await` to sleep for the given number of seconds, which allows all three of these `wait_for` calls to sleep at the same time.

Q1: Understanding Concurrent Execution

The following code starts 3 coroutines that each use the `wait_for` function above to sleep for different amounts of time.

```
import asyncio

async def wait_for(seconds):
    await asyncio.sleep(seconds)

async def go():
    await asyncio.gather(wait_for(5), wait_for(3), wait_for(4))

asyncio.run(go())
```

How many seconds does this code take to run?

5

Implementing a WWPD Race

Q2: Status

Here's a class that can be used to coordinate two different coroutines using a shared object.

The `Status` class keeps track of whether some operation is finished.

The `is_done` method returns a boolean indicating whether the operation that a `Status` represents is done or not. When a `Status` is newly created, `is_done()` returns `False`.

The `done` method indicates that the operation that a `Status` represents has completed. After `done` is called on a `Status`, all future calls to `is_done` return `True`.

```
class Status:
    """A Status represents whether some operation is complete.

    >>> s = Status()
    >>> s.is_done()
    False
    >>> s.done()
    >>> s.is_done()
    True
    """
    def __init__(self):
        ____

    def is_done(self) -> bool:
        return ____

    def done(self):
        ____
```

Q3: Wait for User Input

Implement the coroutine function `get_user_input`, which takes a `Status` instance. It waits for one line of user input, marks the `status` as done, and returns the user input.

```
import sys

async def get_user_input(status: Status) -> str:
    # Read one line of user input
    # Using "await" means that we're giving up control
    # and letting other coroutines run while we wait for
    # user input.
    result = await asyncio.to_thread(sys.stdin.readline)
    ____ # Update status
    return ____
```

Q4: Timer

The coroutine function `timer` takes a `status` and a positive number `period`. It repeatedly prints out how many seconds have elapsed, once every `period` seconds, until it detects that the `status` is done. The first message it prints is `0 seconds have passed...`. While it's waiting to print the next message, it uses `await` so that other coroutines can run concurrently. It returns the number of seconds that passed before it observes that the `status` is done, which should be a multiple of `period`.

When printing that 1 second has passed, use `1 second has passed...` instead of `1 seconds have passed...` to

respect the grammar of English.

For example:

```
async def timer_example():
    """An example of using a timer.

    >>> asyncio.run(timer_example())
    0 seconds have passed...
    0.5 seconds have passed...
    1 second has passed...
    1.5 seconds have passed...
    Status updated after 1.7 seconds
    The timer counted up to 2.0 seconds
    """
    async def update_status(status: Status):
        await asyncio.sleep(1.7)
        print('Status updated after 1.7 seconds')
        status.done()

    status = Status()
    _, elapsed = await asyncio.gather(update_status(status), timer(status, 0.5))
    print(f'The timer counted up to {elapsed} seconds')
```

`await` a call to `asyncio.sleep` within the `while` statement in order to allow other coroutines to run during that time.

```
async def timer(status: Status, period):
    """Print a message every period seconds until status.is_done()."""
    time_passed = 0
    while ____:
        if ____:
            print('1 second has passed...')
        else:
            print(f'{____} seconds have passed...')
        time_passed ____
        ____
    return time_passed
```

Discussion time: Why do we need the `Status` class instead of passing in a `bool` called `status` in order to ensure that the `timer` eventually stops?

Q5: What Would Python Do

`timer` and `get_user_input` can be used together to write a new function `wwpd` that waits for user input while printing out how much time has elapsed every second. `wwpd` works similarly to the "What Would Python Do" questions in lab: it presents the user with a Python expression and waits for the user to enter a response. Unlike lab, our `wwpd` uses `timer` to tell the user how long they have taken to respond.

Your solution will need to assign the name `response` to what the user entered, and `seconds` to the elapsed time returned by `timer`.

Tip: `asyncio.gather` returns a list of the return values of the passed-in coroutines. Values are returned in the same order that the corresponding coroutines were passed into `asyncio.gather`, not the order they finished.

```
async def wwpd(challenge: str):
    """Run a WWPD interface.

    >>> simulate_user_input(5, 2.5) # simulate a user entering 5 after 2.5 seconds
    >>> asyncio.run(wwpd('2 + 3'))
    What is 2 + 3
    0 seconds have passed...
    1 second has passed...
    2 seconds have passed...
    5
    The user waited 3 seconds to correctly respond with 5

    >>> simulate_user_input(6, 1.5) # simulate a user entering 6 after 1.5 seconds
    >>> asyncio.run(wwpd('2 + 3'))
    What is 2 + 3
    0 seconds have passed...
    1 second has passed...
    6
    The user waited 2 seconds to incorrectly respond with 6
    """
    print('What is', challenge)
    "*** YOUR CODE HERE ***"




    # Get the correct answer, and compare it to the user input.
    correct_answer = eval(challenge)
    if str(correct_answer) == response.strip():
        print(f'The user waited {seconds} seconds to correctly respond with {response}')
    else:
        print(f'The user waited {seconds} seconds to incorrectly respond with {response}'
    )

def simulate_user_input(response, time):
    """Simulate a user response after an amount of time. This is just for testing."""
    async def f(status):
        await asyncio.sleep(time)
        status.done()
        print(response)
        return str(response)
    global get_user_input
    get_user_input = f
```

Q6: WWPD Race

Now we'll implement a race to solve WWPD questions faster than an opponent. Implement `run_challenges`, a coroutine function that has a player repeatedly evaluate expressions. It takes the following arguments: * `name: str`: The name of the player * `get_result`: A coroutine function that accepts a Python expression (a `str`) and returns the player's guess of the value of that expression * `expressions: list[str]`: A list of expressions to evaluate * `first: dict[str, str]`: A dictionary mapping expressions to the name of first player that correctly evaluated that expression

`run_challenges` finds the first expression that hasn't yet been correctly evaluated by any player and uses the `get_result` coroutine function to evaluate it. If the expression was evaluated correctly, `run_challenges` updates `first` to contain the name of the player who evaluated the expression correctly first. `run_challenges` continues to try to evaluate expressions that haven't yet been evaluated correctly until there are no un-evaluated expressions left.

After `run_challenges` is called concurrently for 2 players, `first` should contain a mapping of each expression to the first player who correctly evaluated that expression. Keep in mind that a player may start evaluating a particular expression first, but take longer than the other player to reach the correct answer.

Here's an example call to `run_challenge`:

```
def race_example():
    """Run an example with a simulated user input.

    >>> race_example()
    >>> print(asyncio.run(run_challenge("fake human")))
    -- Good job fake human; you correctly evaluated '1 + 1' --
    -- Good job fake human; you correctly evaluated '[1, 2].append([5, 6])' --
    -- Good job computer; you correctly evaluated '1 + 1' --
    -- Not quite fake human. Try to evaluate '[1, 2] + [5, 6]' again! --
    -- Good job computer; you correctly evaluated '[1, 2] + [5, 6]' --
    -- Not quite fake human. Try to evaluate '[1, 2] + [5, 6]' again! --
    {'1 + 1': 'fake human', '[1, 2].append([5, 6])': 'fake human', '[1, 2] + [5, 6]': '
    computer'}
    """
    # Swap this simulated "fake human" response function for the one that gets input
    async def mock_get_input_from_user(expression):
        if expression in ['1 + 1', '[1, 2].append([5, 6])']:
            await asyncio.sleep(0.1)
            return str(eval(expression))
        else:
            await asyncio.sleep(0.2)
            return 'incorrect'
    global get_input_from_user
    get_input_from_user = mock_get_input_from_user
```

You may not need multiple lines in all of the multiline inputs.

It could be fun to copy this code to your laptop to try to see if you can beat the computer! You can work as a group to come up with a new challenge list of Python expressions.

The `first` dictionary might have changed between when `get_next_expression` was called and `result.strip()`

`== correct_answer` was checked because `get_result` is a coroutine (and should be `await`ed). Therefore, make sure to check the contents of `first` again before modifying it rather than assuming something about its contents.

Call `run_challenges` twice from within `run_challenge`, once with `player` as the player and once with `'computer'` as the player. Since those calls should be run concurrently, use `asyncio.gather`, which must be `await`ed to run.

```
def get_next_expression(expressions, first) -> str:
    """ Return the lowest index expression in expressions that's not in first,
        or an empty string if all expressions are in first.

    >>> get_next_expression(['1 + 1', '[1, 2] + [5, 6]'], {})
    '1 + 1'
    >>> get_next_expression(['1 + 1', '[1, 2] + [5, 6]'], {'1 + 1': 'John'})
    '[1, 2] + [5, 6]'
    >>>
    >>> get_next_expression(['1 + 1'], {'1 + 1': 'John'})
    ''
    """
    "*** YOUR CODE HERE ***"









async def run_challenges(name: str, get_result, expressions: list[str], first: dict[str,
    str]):
    expression = get_next_expression(expressions, first)
    while ____:
        result = ____

        correct_answer = str(eval(expression))
        if result.strip() == correct_answer:
            print(f"-- Good job {name}; you correctly evaluated '{expression}' --")
            "*** YOUR CODE HERE ***"




        else:
            print(f"-- Not quite {name}. Try to evaluate '{expression}' again! --")
        "*** YOUR CODE HERE ***"




async def get_input_from_user(expression):
    print('What does the following Python expression evaluate to', expression)
    return await asyncio.to_thread(sys.stdin.readline)

async def get_input_from_computer(expression: str):
    """Return the result of evaluating the expression, after 0.3 seconds."""
    await asyncio.sleep(0.3)
    return str(eval(expression))

async def run_challenge(player: str):
    """ Return a dictionary mapping each expression to the player name that evaluated it
    first.
        One player reads from stdin using the get_input_from_user coroutine and has name
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.