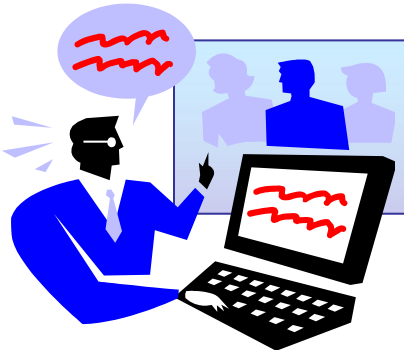
# Asynchronous Processing in Python

olsen software

# Contents


1. Getting started with asynchrony in Python
2. Creating tasks to run in different threads
3. Additional task techniques

Demo folder: **AppxC-Async**

# 1. Getting Started with Asynchrony in Python

- Overview
- Asynchrony in Python
- Coroutines
- The asyncio module
- Simple example of asynchrony

# Overview

- What is asynchrony?
  - The ability to perform multiple tasks concurrently
- Scenarios where asynchrony is important:
  - Processing a large dataset in parallel
  - Handling multiple network connections simultaneously
  - Performing algorithmic processing in the background
  - Etc.

# Asynchrony in Python

- Asynchrony in Python is facilitated by an event loop…
  - The event loop can register tasks to be executed, execute them, delay, or cancel them
- The event loop can optimize I/O
  - If a function is waiting on I/O…
  - The event loop pauses the function and runs another one instead…
  - When the first function completes I/O, it is resumed
- The event loop can also optimize CPU-intensive functions
  - The event loop can pass CPU-intensive functions to a thread pool
  - Thus multiple functions can execute in parallel
- The event loop is encapsulated by an easy-to-use API ☺

# Coroutines

- A coroutine is a special kind of generator function
  - It can cede control during its processing (e.g. for I/O)
  - The event loop then tries to give another coroutine some time
  - The event loop can resume the original coroutine when it's ready

- The preferred way to define a coroutine in modern Python is to prefix a function with the `async` keyword

```
async def someFunc(someArgs) :
    # Some long-running code that might yield control
    #   e.g. code that does slow I/O
    #   e.g. code that CPU-intensive processing
```

# The asyncio Module

- The `asyncio` module provides various methods that allow you to schedule and manage asynchrony
  - Some of the common methods are listed here…

- `asyncio.sleep()`
  - Sleep for a specified delay

- `asyncio.run()`
  - Schedule a coroutine to be run on the current thread

- `asyncio.create_task()`
  - Schedule a coroutine to be run on another thread

# Simple Example of Asynchrony

```
import asyncio
from time import strftime, localtime

async def displayAfter(msg, delay) :
    await asyncio.sleep(delay)
    now = strftime("%H:%M:%S", localtime())
    print("%s %s" % (now, msg))

def main():
    print("*****Start of main*****")
    asyncio.run(displayAfter("Hei", 3))
    asyncio.run(displayAfter("Bye", 5))
    print("*****End of main*****")

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

simpleAsynchrony.py

- `asyncio.sleep()` is a coroutine
- The `await` keyword yields control back to the event loop, which tries to schedule some other work in the meantime
- You can only use the `await` keyword in coroutines, i.e. functions marked as `async`
- You can't just 'invoke' coroutines, you must schedule via `asyncio`

# 2. Creating Tasks to Run in Different Threads

- Overview
- Simple example of creating a task
- Creating and awaiting multiple tasks
- Awaiting multiple tasks to complete

# Overview

- The `asyncio.create_task()` function creates a task
  - The task will run in a different thread
  - The task will be represented by a `Task` object

- The `Task` class has methods that allow you to manage the running of the task, such as:
  - `done()` – has the task completed yet?
  - `cancel()` – stop the task now
  - `result()` – get the result of the task (it must have finished!)

# Simple Example of Creating a Task (1 of 2)

```
from time import strftime, localtime
import asyncio

def doDisplay(msg):
    now = strftime("%H:%M:%S", localtime())
    print("%s %s" % (now, msg))

async def displayAfter(msg, delay) :
    doDisplay("START: " + msg)
    await asyncio.sleep(delay)
    doDisplay("END: " + msg)

async def main():
    print("*****Start of main*****")
    task = asyncio.create_task(displayAfter("Hello", 10))

    for i in range(0,5) :
        print("Doing something useful...")
        await asyncio.sleep(1)

    print("Finished doing useful work, now I'll wait for task to finish")
    await task
    print("*****End of main*****")

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

creatingTask.py

- See the following slide for the output from this code…

# Simple Example of Creating a Task (2 of 2)

- Here's the output for the code on the previous slide

C:\Windows\System32\cmd.exe

```
C:\PythonDev\Demos\AppxC-Async>python creatingTask.py
*****Start of main*****
Doing something useful...
20:20:29 START: Hello
Doing something useful...
Doing something useful...
Doing something useful...
Doing something useful...
Finished doing useful work, now I'll wait for task to finish
20:20:39 END: Hello
*****End of main*****

C:\PythonDev\Demos\AppxC-Async>
```

# Creating Multiple Tasks (1 of 2)

- You can create multiple tasks
  - All the tasks run concurrently
  - So it takes less time overall for everything to complete
  - You can await for each task to complete individually

```
async def main():
    doDisplay("*****Start of main*****")
    task1 = asyncio.create_task(displayAfter("Bonjour", 10))
    task2 = asyncio.create_task(displayAfter("Bore da", 15))
    task3 = asyncio.create_task(displayAfter("Hei hei", 20))

    for i in range(0,5) :
        doDisplay("Doing something useful...")
        await asyncio.sleep(1)

    doDisplay("Waiting for task1 to finish")
    await task1

    doDisplay("Waiting for task2 to finish")
    await task2

    doDisplay("Waiting for task3 to finish")
    await task3

    doDisplay("*****End of main*****")
```

creatingMultipleTasks.py

# Creating Multiple Tasks (2 of 2)

- Here's the output for the code on the previous slide

C:\Windows\System32\cmd.exe

```
C:\PythonDev\Demos\AppxC-Async>python creatingMultipleTasks.py
20:21:20 *****Start of main*****
20:21:20 Doing something useful...
20:21:20 START: Bonjour
20:21:20 START: Bore da
20:21:20 START: Hei hei
20:21:21 Doing something useful...
20:21:22 Doing something useful...
20:21:23 Doing something useful...
20:21:24 Doing something useful...
20:21:25 Waiting for task1 to finish
20:21:30 END: Bonjour
20:21:30 Waiting for task2 to finish
20:21:35 END: Bore da
20:21:35 Waiting for task3 to finish
20:21:40 END: Hei hei
20:21:40 *****End of main*****

C:\PythonDev\Demos\AppxC-Async>
```

# Awaiting Multiple Tasks to Complete (1 of 2)

- The previous example awaited individual tasks to complete
  - If you prefer, to can await multiple tasks to complete
  - Use `asyncio.gather()`, which blocks until all tasks are done

```
async def main():
    doDisplay("*****Start of main*****")
    task1 = asyncio.create_task(displayAfter("Bonjour", 10))
    task2 = asyncio.create_task(displayAfter("Bore da", 15))
    task3 = asyncio.create_task(displayAfter("Hei hei", 20))

    for i in range(0,5) :
        doDisplay("Doing something useful...")
        await asyncio.sleep(1)

    doDisplay("Waiting multiple tasks to complete")
    await asyncio.gather(task1, task2, task3)

    doDisplay("*****End of main*****")
```

awaitingMultipleTasks.py

# Awaiting Multiple Tasks to Complete (2 of 2)

- Here's the output for the code on the previous slide

C:\Windows\System32\cmd.exe

```
C:\PythonDev\Demos\AppxC-Async>python awaitingMultipleTasks.py
20:22:08 *****Start of main*****
20:22:08 Doing something useful...
20:22:08 START: Bonjour
20:22:08 START: Bore da
20:22:08 START: Hei hei
20:22:09 Doing something useful...
20:22:10 Doing something useful...
20:22:11 Doing something useful...
20:22:12 Doing something useful...
20:22:13 Waiting multiple tasks to complete
20:22:18 END: Bonjour
20:22:23 END: Bore da
20:22:28 END: Hei hei
20:22:28 *****End of main*****

C:\PythonDev\Demos\AppxC-Async>
```

# 3. Additional Task Techniques

- Awaiting the result of a task
- Polling a task to see if it's done
- Cancelling a task

# Awaiting the Result of a Task (1 of 2)

- A coroutine can return a value
  - The calling code would like to retrieve the value when complete

- Here's one way for the calling code to do this:
  - Create a task, to schedule the coroutine in a separate thread
  - Await completion of the task
  - The `await` expression gives the result of the completed coroutine

```
async def createStringAfter(msg, delay) :
    await asyncio.sleep(delay)
    now = strftime("%H:%M:%S", localtime())
    return "{0} {1}".format(now, msg)

async def main():
    print("*****Start of main*****")
    task = asyncio.create_task(createStringAfter("Bonjour", 10))
    result = await task
    print(result)
    print("*****End of main*****")
```

awaitingTaskResult1.py

# Awaiting the Result of a Task (2 of 2)

- The previous slide created a task, and then awaited its completion separately:

```
async def main():
    print("*****Start of main*****")
    task = asyncio.create_task(createStringAfter("Bonjour", 10))
    result = await task
    print(result)
    print("*****End of main*****")
```

awaitingTaskResult1.py

- If it's more convenient, you can combine these two statements into a single statement

```
async def main():
    print("*****Start of main*****")
    result = await asyncio.create_task(createStringAfter("Bonjour", 10))
    print(result)
    print("*****End of main*****")
```

awaitingTaskResult2.py

# Polling a Task to See if it's Done

- Sometimes you might want to poll a task to see if it's done
  - Call `done()` on the task, to see if it's finished
  - If it hasn't finished, do something else for a bit, then check again
  - When it really has finished, call `result()` on the task

```
async def main():
    doDisplay("*****Start of main*****")
    task = asyncio.create_task(createStringAfter("Bonjour", 10))

    while True:
        if task.done():
            result = task.result()
            doDisplay(result)
            break
        else:
            doDisplay("Doing something useful...")
            await asyncio.sleep(1)

    doDisplay("*****End of main*****")
```

pollingTask.py

# Cancelling a Task

- Sometimes you might want to cancel a task mid-flight
  - Call `cancel()` on the task

```
async def main():
    doDisplay("*****Start of main*****")
    task = asyncio.create_task(createStringAfter("Bonjour", 10))

    while True:
        if task.done():
            result = task.result()
            doDisplay(result)
            break
        else:
            cancel = input("Task not complete yet. Do you want to cancel it? ")
            if cancel == "y":
                doDisplay("OK I'll cancel the task and we'll all just move on in life.")
                task.cancel()
                break
            else:
                doDisplay("OK I'll wait another second and do something useful...")
                await asyncio.sleep(1)

    doDisplay("*****End of main*****")
```

cancellingTask.py

# Any Questions?