async and await



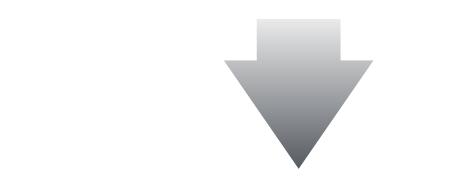
Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

- Coroutines and asynchronous code
- Runtime support requirements

Coroutines and Asynchronous Code

in a more natural style

```
fn read_exact<T: Read>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
   let mut cursor = 0;
   while cursor < buf.len() {</pre>
        cursor += input.read(&mut buf[cursor..])?;
```



```
async fn read_exact<T: AsyncRead>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
   let mut cursor = 0;
   while cursor < buf.len() {</pre>
        cursor += input.read(&mut buf[cursor..]).await?;
```

await calls \rightarrow low-overhead concurrent I/O



Aims to provide language and run-time support for I/O multiplexing on a single thread,

• Runtime schedules async functions on a thread pool, yielding to other code on

 Structure I/O-based code as a set of c sources and yield in place of blocking

What is a coroutine?

A generator **yield**s a sequence of values:

A function that can repeatedly run, yielding a sequence of values, while maintaining internal state

Calling **countdown(5)** produces a *generator object*. The **for** loop protocol calls **next()** on that object, causing it to execute until the next **yield** statement and return the yielded value.

 \rightarrow Heap allocated; maintains state; executes only in response to external stimulus



Structure I/O-based code as a set of concurrent coroutines that accept data from I/O

Based on: http://www.dabeaz.com/coroutines/Coroutines.pdf

 Structure I/O-based code as a set of c sources and yield in place of blocking

What is a coroutine?

A coroutine more generally consumes and yields values:

```
def grep(pattern):
    print(F"Looking for {pattern}")
    while True:
        line = (yield)
        if pattern in line:
        print line
        CC
>>> g = grep("python")
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("Python generators rock!")
python generators rock!
```



Structure I/O-based code as a set of concurrent coroutines that accept data from I/O

The coroutines executes in response to **next()** or **send()** calls

Calls to **next()** make it execute until it next call **yield** to return a value

Calls to **send()** pass a value into the coroutine, to be returned by **(yield)**

Based on: <u>http://www.dabeaz.com/coroutines/Coroutines.pdf</u>

 Structure I/O-based code as a set of c sources and yield in place of blocking

What is a coroutine?

A coroutine is a function that executes *concurrently* to – but not in parallel with – the rest of the code

It is event driven, and can accept and return values



Structure I/O-based code as a set of concurrent coroutines that accept data from I/O

- sources and yield in place of blocking
 - An **async** function is a coroutine
 - coroutine while the I/O is performed
 - Provides concurrency without parallelism
 - Coroutines operate concurrently, but typically within a single thread
 - await passes control to another coroutine, and schedules a later wake-up for when the awaited operation completes
 - Encodes down to a state machine with calls to **select()**, or similar ullet
 - Mimics structure of code with multi-threaded I/O within a single thread



Structure I/O-based code as a set of concurrent coroutines that accept data from I/O

Blocking I/O operations are labelled in the code – await – and cause control to pass to another

async Functions

- An **async** function is one that can act
 - It is executed asynchronously by the run
 - Widely supported Python 3, JavaScrip

```
#!/usr/bin/env python3
import asyncio
async def fetch_html(url: str, session: Client
    resp = await session.request(method="GET",
    html = await resp.text()
    return html
...
```

• Main program must trigger asynchronous execution by the runtime:

```
asyncio.run(async function)
```

- Starts asynchronous polling runtime, runs until specified **async** function completes
- Runtime drives **async** functions to completion and handles switching between coroutines



t as a coroutine	
ntime	
ot, C#, Rust,	
Session) -> str: url=url)	async tag on function
	$\texttt{yield} \rightarrow \texttt{await}$
	But essentially a coroutine

await Future Results

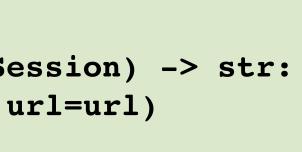
- An await operation yields from the coroutine

 - Triggers I/O operation and adds corresponding file descriptor to set polled by the runtime • Puts the coroutine in queue to be woken by the runtime, when file descriptor becomes ready

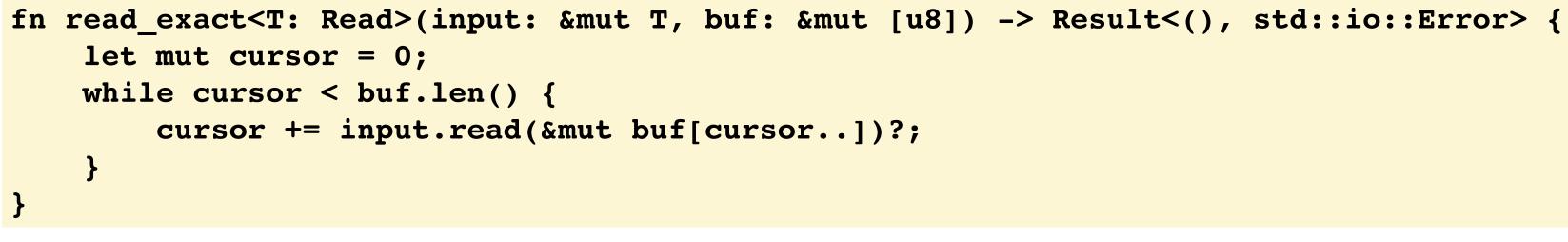
```
#!/usr/bin/env python3
import asyncio
async def fetch_html(url: str, session: ClientSession) -> str:
    resp = await session.request(method="GET", url=url)
    html = await resp.text()
    return html
...
```

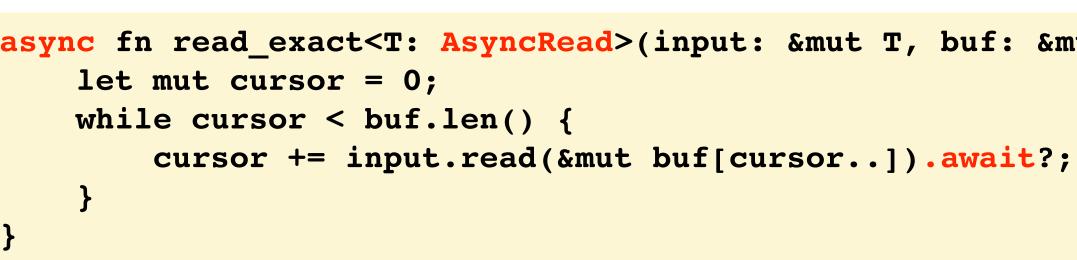
- If another coroutine is ready to execute then schedule wake-up once the I/O completes, and pass control passes to the other coroutine; else runtime blocks until either this, or some other, I/O operation becomes ready
- At some later time the file descriptor becomes ready and the runtime reschedules the coroutine – the I/O completes and the execution continues





async and await programming model





- Annotations (async, await) indicate asynchrony, context switch points
 - when I/O operations occur



• Resulting asynchronous code should follow structure of synchronous (blocking) code:

```
async fn read_exact<T: AsyncRead>(input: &mut T, buf: &mut [u8]) -> Result<(), std::io::Error> {
```

Compiler and runtime work together to generate code that can be executed in fragments

Runtime Support

- sources for activity
- An **async** function that returns data of type **T** compiles to a regular function that returns impl Future<Output=T>

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, lw: &LocalWaker) -> Poll<Self::Output>;
}
```

- The runtime continually calls **poll()** on **Future** values until all are **Ready**
 - A future returns **Ready** when complete, **Pending** when blocked on **await**ing I/O
 - Calling tokio::run(future) starts the runtime
- Well supported in Python and JavaScript runtime for Rust is experimental: <u>https://tokio.rs/</u>



Asynchronous code needs runtime support to execute the coroutines and poll the I/O



pub enum Poll<T> { Ready(T), Pending,

• i.e., it returns a Future value that represents a value that will become available later

async and await



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

- Coroutines and asynchronous code
- Runtime support requirements