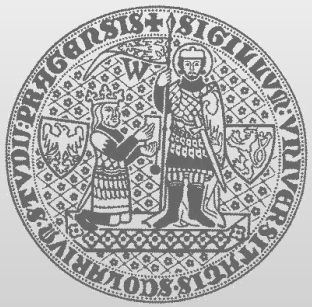
NPRG065: Programming in Python
Lecture 12

http://d3s.mff.cuni.cz

Tomas Bures

Petr Hnetynka

{bures,hnetynka}@d3s.mff.cuni.cz



CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Coroutines

- Asynchronous functions
  - typically for I/O or long computations
  - asynchronous = not waiting for a result
  - in Python – coroutines

- Coroutine function – function defined via **`async def`** or decorated with **`@asyncio.coroutine`**
  - **`async def`** preferred

- Coroutine object – obtained by calling a coroutine function
- Calling a coroutine does not start its code running

Coroutines, tasks, futures

- Task – associates coroutine with an event loop (i.e. thread of execution)
  - Itself is an awaitable object

- Task is also a Future
  - Future provides callback interface to register for results
    - add_done_callback, remove_done_callback
    - cancelled, done, result
  - Future acts as a bridge between coroutines and callback-based internal methods (e.g. for manipulating streams)

Coroutines

- Things a coroutine can do:
  - **`result = await future`** or **`result = yield from future`**
    - suspends the coroutine until the future is done, then returns the future's result
  - **`result = await coroutine`** or **`result = yield from coroutine`**
    - wait for another coroutine to produce a result
  - **`return expression`**
  - **`raise exception`**

Coroutines

- Calling a coroutine – returns a coroutine object
- To start a coroutine
  - call **`await coroutine`** or **`yield from coroutine`** from *another coroutine*, or
  - schedule its execution via *the even loop*
    - **`asyncio.get_event_loop()`**
- Event loop and related methods (the mostly used ones)
  - **`asyncio.ensure_task(coroutine)`**
    - schedules the execution of a coroutine object, wraps it in a future
  - **`loop.run_until_complete(coroutine)`**
    - runs until the task is done
      - if the argument is a coroutine object, it is wrapped by **`ensure_task()`**
- Future – encapsulates a call ~ a place for result
  - **`cancel() -> bool`**
  - **`canceled() -> bool`**
  - **`done() -> bool`**
  - **`result()`**
  - **`exception()`**

Examine and run
01_co_hello.py
02_co_print_time.py

Coroutines

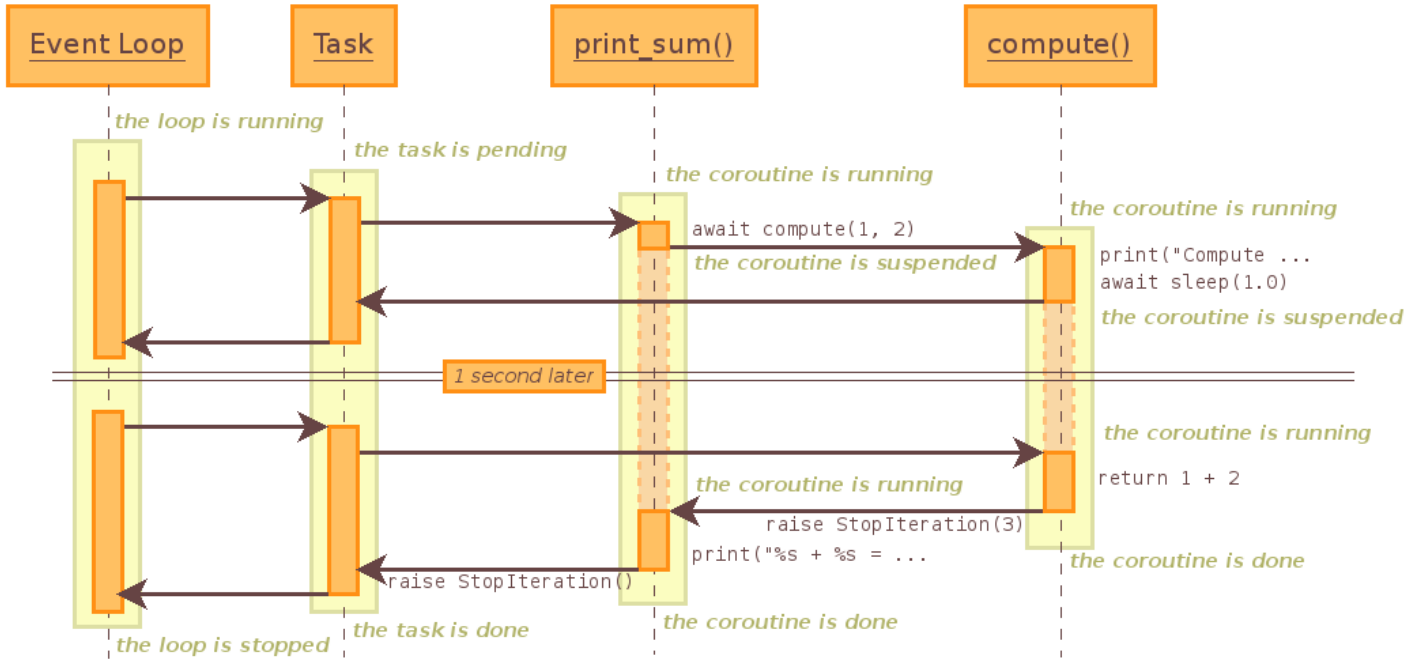
```
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y

async def print_sum(x, y):
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.ge
loop.run_until_co
loop.close()
```

Code in
03_co_sum.py



Coroutines

- If coroutine waits for another coroutine, it is suspended -> other coroutine can run
- -> synchronous waiting in coroutine blocks other coroutines

Compare 04a_co_sleep.py and 04b_co_sleep.py

- -> for async I/O operations – we need async aware functions
  - e.g. **`urllib.request`** module is not async aware
    -> use **`aiohttp`** (but it is not in the std library)

Compare 05a_co_down.py and 05b_co_down.py

- **`async with`** and **`async for`**
  - like regular **`with`** and **`for`** but can yield

Coroutines under the Hood

- Coroutine is an awaitable object
  - Works in a similar way to a generator

- Coroutine
  - Started by __await__
  - Continuation controlled by methods “send”, “throw”, “close”
- Generator
  - Started by __next__
  - Continuation controlled by methods “send”, “throw”, “close”

The slides are licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.