17.4. `concurrent.futures` — Launching parallel tasks

New in version 3.2.

Source code: Lib/concurrent/futures/thread.py and Lib/concurrent/futures/process.py

The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using `ThreadPoolExecutor`, or separate processes, using `ProcessPoolExecutor`. Both implement the same interface, which is defined by the abstract `Executor` class.

17.4.1. Executor Objects

class `concurrent.futures.`**`Executor`**

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

`submit`(*fn*, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as `fn(*args **kwargs)` and returns a `Future` object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

`map`(*func*, **iterables*, *timeout=None*, *chunksize=1*)

Similar to `map(func, *iterables)` except:

- the *iterables* are collected immediately rather than lazily;
- *func* is executed asynchronously and several calls to *func* may be made concurrently.

The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `Executor.map()`. *timeout* can be an int or a float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If a *func* call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using `ProcessPoolExecutor`, this method chops *iterables* into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting *chunksize* to a positive integer. For very long iterables, using a large value for *chunksize* can significantly improve performance compared to the default size of 1. With `ThreadPoolExecutor`, *chunksize* has no effect.

Changed in version 3.5: Added the *chunksize* argument.

shutdown(*wait=True*)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to `Executor.submit()` and `Executor.map()` made after shutdown will raise `RuntimeError`.

If *wait* is `True` then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If *wait* is `False` then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of *wait*, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the `with` statement, which will shutdown the `Executor` (waiting as if `Executor.shutdown()` were called with *wait* set to `True`):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.4.2. ThreadPoolExecutor

`ThreadPoolExecutor` is an `Executor` subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a `Future` waits on the results of another `Future`. For example:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting c
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting c
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class `concurrent.futures.`**`ThreadPoolExecutor`**(*max_workers=None*, *thread_name_prefix=''*)

An `Executor` subclass that uses a pool of at most *max_workers* threads to execute calls asynchronously.

Changed in version 3.5: If *max_workers* is `None` or not given, it will default to the number of processors on the machine, multiplied by `5`, assuming that `ThreadPoolExecutor` is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for `ProcessPoolExecutor`.

New in version 3.6: The *thread_name_prefix* argument was added to allow users to control the threading.Thread names for worker threads created by the pool for easier debugging.

17.4.2.1. ThreadPoolExecutor Example

```
import concurrent.futures
import urllib.request
```

```
URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptl
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url i
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.4.3. ProcessPoolExecutor

The `ProcessPoolExecutor` class is an `Executor` subclass that uses a pool of processes to execute calls asynchronously. `ProcessPoolExecutor` uses the `multiprocessing` module, which allows it to side-step the Global Interpreter Lock but also means that only picklable objects can be executed and returned.

The `__main__` module must be importable by worker subprocesses. This means that `ProcessPoolExecutor` will not work in the interactive interpreter.

Calling `Executor` or `Future` methods from a callable submitted to a `ProcessPoolExecutor` will result in deadlock.

class `concurrent.futures.`**`ProcessPoolExecutor`**(*max_workers=None*)

An `Executor` subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is `None` or not given, it will default to the number of processors on the machine. If *max_workers* is lower or equal to `0`, then a `ValueError` will be raised.

Changed in version 3.3: When one of the worker processes terminates abruptly, a `BrokenProcessPool` error is now raised. Previously, behaviour was undefined but operations on the executor or its futures would often freeze or deadlock.

17.4.3.1. ProcessPoolExecutor Example

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES
            print('%d is prime: %s' % (number, prime))

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

17.4.4. Future Objects

The `Future` class encapsulates the asynchronous execution of a callable. `Future` instances are created by `Executor.submit()`.

class `concurrent.futures.`**`Future`**

Encapsulates the asynchronous execution of a callable. `Future` instances are created by `Executor.submit()` and should not be created directly except for testing.

`cancel`()

Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()

Return `True` if the call was successfully cancelled.

running()

Return `True` if the call is currently being executed and cannot be cancelled.

done()

Return `True` if the call was successfully cancelled or finished running.

result(*timeout=None*)

Return the value returned by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call raised, this method will raise the same exception.

exception(*timeout=None*)

Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to *timeout* seconds. If the call hasn't completed in *timeout* seconds, then a `concurrent.futures.TimeoutError` will be raised. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

If the future is cancelled before completing then `CancelledError` will be raised.

If the call completed without raising, `None` is returned.

add_done_callback(*fn*)

Attaches the callable *fn* to the future. *fn* will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added and are always called in a thread belonging to the process that add-

ed them. If the callable raises an `Exception` subclass, it will be logged and ignored. If the callable raises a `BaseException` subclass, the behavior is undefined.

If the future has already completed or been cancelled, *fn* will be called immediately.

The following `Future` methods are meant for use in unit tests and `Executor` implementations.

`set_running_or_notify_cancel`()

This method should only be called by `Executor` implementations before executing the work associated with the `Future` and by unit tests.

If the method returns `False` then the `Future` was cancelled, i.e. `Future.cancel()` was called and returned *True*. Any threads waiting on the `Future` completing (i.e. through `as_completed()` or `wait()`) will be woken up.

If the method returns `True` then the `Future` was not cancelled and has been put in the running state, i.e. calls to `Future.running()` will return *True*.

This method can only be called once and cannot be called after `Future.set_result()` or `Future.set_exception()` have been called.

`set_result`(*result*)

Sets the result of the work associated with the `Future` to *result*.

This method should only be used by `Executor` implementations and unit tests.

`set_exception`(*exception*)

Sets the result of the work associated with the `Future` to the `Exception` *exception*.

This method should only be used by `Executor` implementations and unit tests.

17.4.5. Module Functions

`concurrent.futures.`**`wait`**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the `Future` instances (possibly created by different `Executor` instances) given by *fs* to complete. Returns a named 2-tuple of sets. The first set, named `done`, contains the futures that completed (finished or were cancelled) before the wait completed. The second set, named `not_done`, contains uncompleted futures.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants:

Constant	Description
`FIRST_COMPLETED`	The function will return when any future finishes or is cancelled.
`FIRST_EXCEPTION`	The function will return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to `ALL_COMPLETED`.
`ALL_COMPLETED`	The function will return when all futures finish or are cancelled.

`concurrent.futures.`**`as_completed`**(*fs*, *timeout=None*)

Returns an iterator over the `Future` instances (possibly created by different `Executor` instances) given by *fs* that yields futures as they complete (finished or were cancelled). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before `as_completed()` is called will be yielded first. The returned iterator raises a `concurrent.futures.TimeoutError` if `__next__()` is called and the result isn't available after *timeout* seconds from the original call to `as_completed()`. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

See also:

PEP 3148 – futures - execute computations asynchronously

The proposal which described this feature for inclusion in the Python standard library.

17.4.6. Exception classes

exception `concurrent.futures.`**`CancelledError`**

Raised when a future is cancelled.

exception `concurrent.futures.`**`TimeoutError`**

Raised when a future operation exceeds the given timeout.

exception `concurrent.futures.process.`**`BrokenProcessPool`**

Derived from `RuntimeError`, this exception class is raised when one of the workers of a `ProcessPoolExecutor` has terminated in a non-clean fashion (for example, if it was killed from the outside).

New in version 3.3.