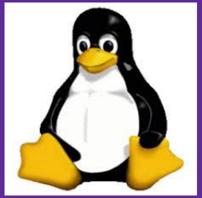
Introduction to Computation

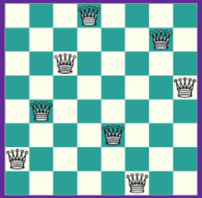
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

chengfan85@gmail.com
https://github.com/ichengfan/itc





15

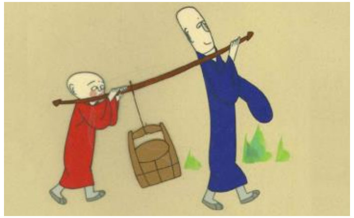
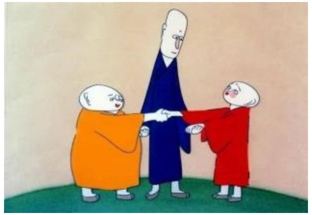
Outline

- Threading
- Multiprocessing
- Async IO
- Regular Expression

Real-world Computation: Multitask


Human — Listen, Speak, Read, Write, Touch


Quantum Mechanics—Parallel universe

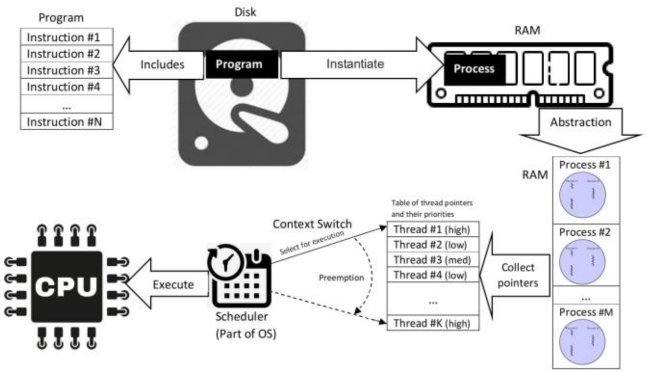




How to coordinate three monks


CPU — Intel Pentium 4 (2000, single-core CPUs)
2018 Core i9-9980X 18 Cores (36 Thread)

从CPU的角度看问题

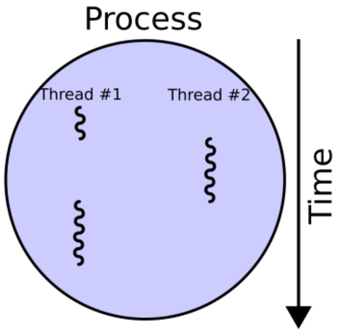
Process (线程)



- In windows, each *.exe file will be linked with a process — Wechat, Word, VS Code, Chrome
- While a computer program is a passive collection of instructions typically stored in a file on disk, a process is the execution of those instructions after being loaded from the disk into memory

- The operating system keeps its processes separate and allocates the resources they need, so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing).
  - The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways
- In computing, a process (进程) is being executed by one or many threads (线程), which comprises the program code, assigned system resources, physical and logical access permissions, and data structures to initiate, control and coordinate execution activity

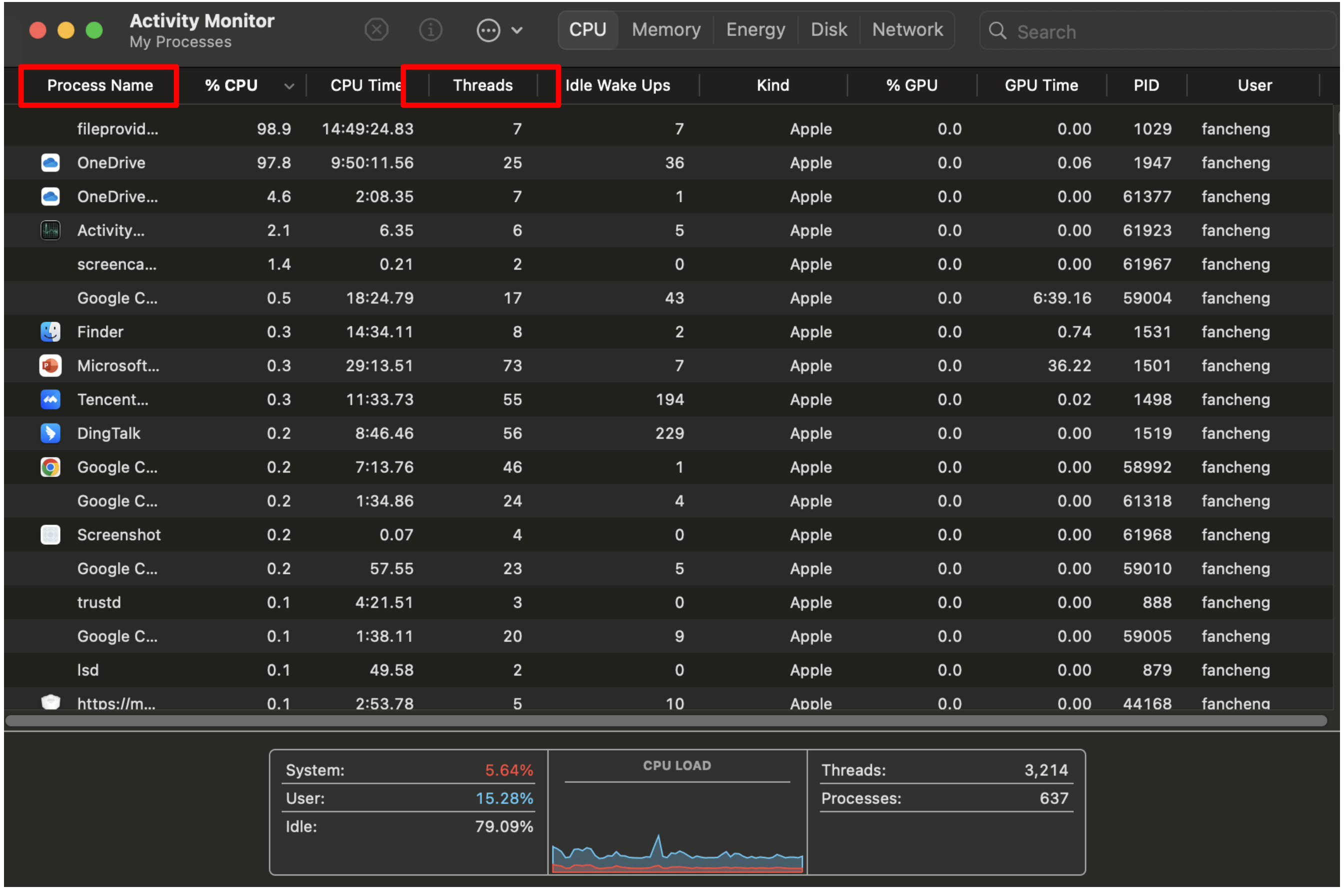
Thread (进程)



- Thread is a lightweight Process
- OS: 政府公共服务，Process：公司， Thread：分公司
- 进程、线程竞争：软硬件资源
- In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system
    - In many cases, a thread is a component of a process
- The multiple threads of a given process may be executed concurrently (via multithreading capabilities), sharing resources such as memory, while different processes do not share these resources.
    - In particular, the threads of a process share its executable code and the values of its dynamically allocated variables and non-thread-local global variables at any given time

Process and Thread

Process: 公司
Thread: 员工



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	Kind	% GPU	GPU Time	PID	User
fileprovid...	98.9	14:49:24.83	7	7	Apple	0.0	0.00	1029	fancheng
OneDrive	97.8	9:50:11.56	25	36	Apple	0.0	0.06	1947	fancheng
OneDrive...	4.6	2:08.35	7	1	Apple	0.0	0.00	61377	fancheng
Activity...	2.1	6.35	6	5	Apple	0.0	0.00	61923	fancheng
screenca...	1.4	0.21	2	0	Apple	0.0	0.00	61967	fancheng
Google C...	0.5	18:24.79	17	43	Apple	0.0	6:39.16	59004	fancheng
Finder	0.3	14:34.11	8	2	Apple	0.0	0.74	1531	fancheng
Microsoft...	0.3	29:13.51	73	7	Apple	0.0	36.22	1501	fancheng
Tencent...	0.3	11:33.73	55	194	Apple	0.0	0.02	1498	fancheng
DingTalk	0.2	8:46.46	56	229	Apple	0.0	0.00	1519	fancheng
Google C...	0.2	7:13.76	46	1	Apple	0.0	0.00	58992	fancheng
Google C...	0.2	1:34.86	24	4	Apple	0.0	0.00	61318	fancheng
Screenshot	0.2	0.07	4	0	Apple	0.0	0.00	61968	fancheng
Google C...	0.2	57.55	23	5	Apple	0.0	0.00	59010	fancheng
trustd	0.1	4:21.51	3	0	Apple	0.0	0.00	888	fancheng
Google C...	0.1	1:38.11	20	9	Apple	0.0	0.00	59005	fancheng
lsd	0.1	49.58	2	0	Apple	0.0	0.00	879	fancheng
https://m...	0.1	2:53.78	5	10	Apple	0.0	0.00	44168	fancheng

System:	5.64%	CPU LOAD	Threads:	3,214
User:	15.28%		Processes:	637
Idle:	79.09%			

Blocking function (阻塞函数)

场景分析：食堂，大家排队买午餐，服务员按顺序给每位同学服务。第一位同学（同学A）没有带饭卡（手机没电等等）

- 措施（一）：服务员等待同学A把饭卡找到(可能要10分钟)，付款买午饭，然后服务下一位同学
- 措施（二）：服务员让该同学A到队伍旁边处理饭卡问题，然后服务下一位同学，等同学A找到饭卡后，再招待A

- 程序的运行并仅仅仅仅取决于程序员和CPU，还取决于外界的状态
  - 程序等待网络端传输信号
  - 邮件客户端，收邮件
  - 文件读写，文件被占用
- In computing, a process always exists in exactly one process state. A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation
- 阻塞的情况，并不是CPU不执行程序，而是程序要等待外面的资源，而且不让出CPU使用权，从而导致CPU时间被浪费
- Process/Thread: CPU会在不同的P/T间进行切换和调度，让不同的程序都可以被执行，从而提高CPU利用率
  - 饭店：服务员在不同餐桌间轮流服务
  - 即使单个CPU，多个Thread切换也可以改善性能
  - Process/Thread不会减少总的计算量，但是可以利用阻塞的CPU时间，从而提高效率

Threading

threading

A thread is a separate flow of execution

- This means that your program will have two things happening at once
- This module constructs higher-level threading interfaces on top of the lower level _thread module
  - Changed in version 3.7: This module used to be optional, it is now always available
- The design of this module is loosely based on Java's threading model
  - However, where Java makes locks and condition variables basic behavior of every object, they are separate objects in Python
  - Python's Thread class supports a subset of the behavior of Java's Thread class; currently, there are no priorities, no thread groups, and threads cannot be destroyed, stopped, suspended, resumed, or interrupted
  - The static methods of Java's Thread class, when implemented, are mapped to module-level functions
  - In the Python 2.x series, this module contained camelCase names for some methods and functions. These are deprecated as of Python 3.10, but they are still supported for compatibility with Python 2.5 and lower
- All of the methods described below are executed atomically
- Because of the way CPython implementation of Python works, threading may not speed up all tasks. This is due to interactions with the GIL that essentially limit one Python thread to run at a time

https://docs.python.org/3/library/threading.html

Thread Object

- The Thread class represents an activity that is run in a separate thread of control
- There are two ways to specify the activity:
  - by passing a callable object to the constructor
  - or by overriding the run() method in a subclass
  - No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the __init__() and run() methods of this class
- Once a thread object is created, its activity must be started by calling the thread's start() method. This invokes the run() method in a separate thread of control
- Once the thread's activity is started, the thread is considered 'alive'. It stops being alive when its run() method terminates – either normally, or by raising an unhandled exception. The is_alive() method tests whether the thread is alive
- Other threads can call a thread's join() method. This blocks the calling thread until the thread whose join() method is called is terminated
- A thread has a name. The name can be passed to the constructor, and read or changed through the name attribute
- If the run() method raises an exception, threading.excepthook() is called to handle it. By default, threading.excepthook() ignores silently SystemExit

threading.Thread

class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)

- This constructor should always be called with keyword arguments. Arguments are:
    - group should be None; reserved for future extension when a ThreadGroup class is implemented
    - target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called
    - name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number, or “Thread-N (target)” where “target” is target.__name__ if the target argument is specified
    - args is a list or tuple of arguments for the target invocation. Defaults to ()
    - kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}
- If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread

threading.Thread: Example

- Thread: 必须用关键字传递参数
- args必须是tuple或者list
- start()函数启动线程

```python
import threading

t1 = threading.Thread(target=print, args=("hello thread 1",))
t2 = threading.Thread(target=print, args="hello thread 2")

t1.start()
t2.start()
```

```
hello thread 1
h e l l o   t h r e a d   2
```

```python
import threading

t1 = threading.Thread(target=print, args=("hello thread 3",))
t2 = threading.Thread(target=print, args="hello thread 4")

t2.start()
t1.start()
```

```
h hello thread 3e l l o
 t h r e a d   4
```

- target可以是任意函数
- 线程之间的运行顺序不确定

```
hello thread
hello thread
12
2251626268912
<class 'str'>
```

```python
import threading


message = "hello thread"
t1 = threading.Thread(target=print, args=(message,))
t1.start()


def my_print(message):
    print(message)
    for f in (len, id, type):
        print(f(message))


t1 = threading.Thread(target=my_print, args=(message,))
t1.start()
```

start() and run()

This (start()) invokes the run() method in a separate thread of control

- start()—Start the thread's activity. It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control
- This method will raise a RuntimeError if called more than once on the same thread object

```python
import threading

class MyThread(threading.Thread):
    def __init__(self, *args, **kwargs):
        super(MyThread, self).__init__(*args, **kwargs)

    def run(self):
        print("called by threading.Thread.start()")

if __name__ == '__main__':
    mythread = MyThread()
    mythread.start()
    mythread.join()
```

```
called by threading.Thread.start()
```

https://stackoverflow.com/questions/23224784/start-vs-run-for-threads-in-python

Daemon Thread (守护线程)

- The main program will terminate only after there is no alive non-daemon thread
- A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the daemon property or the daemon constructor argument
- If not None, daemon explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread
- Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an Event
- There is a “main thread” object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread
- Daemons are only useful when the main program is running, and it's okay to kill them off once the other non-daemon threads have exited. Without daemon threads, we have to keep track of them, and tell them to exit, before our program can completely quit. By setting them as daemon threads, we can let them run and forget about them, and when our program quits, any daemon threads are killed automatically

Daemon Thread: Example

- Usually our main program implicitly waits until all other threads have completed their work. However, sometimes programs spawn a thread as a daemon that runs without blocking the main program from exiting. Using daemon threads is useful for services where there may not be an easy way to interrupt the thread or where letting the thread die in the middle of its work without losing or corrupting data

```python
import threading
import time


def my_print(*args):
    print("Enter my print")
    print(args)
    time.sleep(1)
    print("Leave my print")


if __name__ == "__main__":
    print("Enter main")

    dm_thread1 = threading.Thread(target=my_print, args=("hello Daemon",), daemon=None)
    dm_thread1.start()

    print("Quit main")
```

```
Enter main
Enter my print
Quit main
('hello Daemon',)
Leave my print
```

```python
import threading
import time


def my_print(*args):
    print("Enter my print")
    print(args)
    time.sleep(1)
    print("Leave my print")


if __name__ == "__main__":
    print("Enter main")

    dm_thread1 = threading.Thread(target=my_print, args=("hello Daemon",), daemon=True)
    dm_thread1.start()

    print("Quit main")
```

```
Enter main
Enter my printQuit main

('hello Daemon',)
```

join()

join(timeout=None): Wait until the thread terminates

- This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs
- When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out
- When the timeout argument is not present or None, the operation will block until the thread terminates
- A thread can be join()ed many times
- join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raise the same exception

join(): Example

- If you want to be able to join a thread, it's better to not make it a daemon. Daemon threads are for when you want a thread to do its thing and you're not too concerned about when or if it finishes
- The point of making daemon threads is that the program will exit when there are no non-daemon threads left alive

```python
import threading
import time


def my_print1(*args):
    print("Enter my print1")
    print(args)
    time.sleep(1)
    print("Leave my print1")

def my_print2(*args):
    print("Enter my print2")
    print(args)
    time.sleep(1)
    print("Leave my print2")

t1 = threading.Thread(target=my_print1, args=("hello thread 1",))
t2 = threading.Thread(target=my_print2, args="hello thread 2")

t1.start()
t2.start()

t2.join()
t1.join()

print("Quit main.")
```

```
Enter my print1
Enter my print2('hello thread 1',)

('h', 'e', 'l', 'l', 'o', ' ', 't', 'h', 'r', 'e', 'a', 'd', ' ', '2')
Leave my print1
Leave my print2
Quit main.
```

Lock

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked.

- In Python, it is currently the lowest level synchronization primitive available, implemented directly by the _thread extension module
- A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, acquire() and release(). When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, then the acquire() call resets it to locked and returns. The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.
- Locks also support the context management protocol
- When more than one thread is blocked in acquire() waiting for the state to turn to unlocked, only one thread proceeds when a release() call resets the state to unlocked; which one of the waiting threads proceeds is not defined and may vary across implementations
- All methods are executed atomically

Lock: Example

```python
import threading
import time


def print_to_console(msg, lock):
    lock.acquire()
    print(f'Enter {msg}')
    time.sleep(1)
    print(f'Leave {msg}')
    lock.release()


msg = [f"hello thread {x}" for x in range(1, 4)]
lock = threading.Lock()

t1 = threading.Thread(target=print_to_console, args=(msg[0], lock))
t2 = threading.Thread(target=print_to_console, args=(msg[1], lock))
t3 = threading.Thread(target=print_to_console, args=(msg[2], lock))

t3.start()
t2.start()
t1.start()
```

```
Enter hello thread 3
Leave hello thread 3
Enter hello thread 2
Leave hello thread 2
Enter hello thread 1
Leave hello thread 1
```

死锁：一双筷子，两位同学都要用来吃饭；但是一人一只，谁都没法吃饭

- acquire(blocking=True, timeout=- 1)
  - Acquire a lock, blocking or non-blocking
- release()
  - Release a lock. This can be called from any thread, not only the thread which has acquired the lock
- locked()
  - Return True if the lock is acquired

- Two Threads: T1, T2
  Two Locks: L1, L2
- T1 owns L1
  T2 owns L2
- T1 wait for L2
  T2 wait for L1
- Dead lock between T1 and T2
- Solution: Thread Synchronization

CPU Bound and IO Bound

- When you look at a typical Python program—or any computer program for that matter—there's a difference between those that are CPU-bound in their performance and those that are I/O-bound
- CPU-bound (CPU瓶颈) programs are those that are pushing the CPU to its limit. This includes programs that do mathematical computations like matrix multiplications, searching, image processing, etc
- I/O-bound (IO瓶颈) programs are the ones that spend time waiting for Input/Output which can come from a user, file, database, network, etc. I/O-bound programs sometimes have to wait for a significant amount of time till they get what they need from the source due to the fact that the source may need to do its own processing before the input/output is ready, for example, a user thinking about what to enter into an input prompt or a database query running in its own process

GIL: global interpreter lock

- The mechanism used by the CPython interpreter to assure that only one thread executes Python bytecode at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines
- In the words of Larry Hastings, the design decision of the GIL is one of the things that made Python as popular as it is today
- Python has been around since the days when operating systems did not have a concept of threads. Python was designed to be easy-to-use in order to make development quicker and more and more developers started using it
- The creator and BDFL of Python, Guido van Rossum, gave an answer to the community in September 2007 in his article “It isn’t Easy to remove the GIL”:
  - “I’d welcome a set of patches into Py3k only if the performance for a single-threaded program (and for a multi-threaded but I/O-bound program) does not decrease”

https://realpython.com/python-gil/

GIL Remove: Python 3.12+

- A Per-Interpreter GIL
- PEP 684 introduces a per-interpreter GIL, so that sub-interpreters may now be created with a unique GIL per interpreter. This allows Python programs to take full advantage of multiple CPU cores. This is currently only available through the C-API, though a Python API is anticipated for 3.13
- The main focus of PEP 684 is refactoring the internals of the CPython source code so that each subinterpreter can have its own global interpreter lock (GIL). The GIL is a lock, or mutex, which allows only one thread to have control of the Python interpreter. Until this PEP, there was a single GIL for all subinterpreters, which meant that no matter how many subinterpreters you created, only one could run at a single time
- Moving the GIL so that each subinterpreter has a separate lock is a great idea. So, why hasn't it been done already? The issue is that the GIL is preventing multiple threads from accessing some of the global state of CPython simultaneously, so it's protecting your program from bugs that race conditions could cause

https://realpython.com/python312-subinterpreters/#changes-to-the-global-state-and-the-gil

Multiprocessing

Multiprocessing

- multiprocessing is a package that supports spawning processes using an API similar to the threading module.
- The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads.
- Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both POSIX and Windows.
- In multiprocessing, processes are spawned by creating a Process object and then calling its start() method. Process follows the API of threading.Thread

Due to GIL, multiprocessing is the real parallel computing.
Multithreading will remove GIL in Python 3.12+

https://docs.python.org/3/library/multiprocessing.html

Contexts and start methods

仅供了解

- Thread 会与主程序共享代码和数据，Process不一定
- Depending on the platform, multiprocessing supports three ways to start a process. These start methods are
- spawn
    - The parent process starts a fresh Python interpreter process. The child process will only inherit those resources necessary to run the process object's run() method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using fork or forkserver
    - Available on POSIX and Windows platforms. The default on Windows and macOS
- fork
    - The parent process uses os.fork() to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic
    - Available on POSIX systems. Currently the default on POSIX except macOS
    - Note The default start method will change away from fork in Python 3.14. Code that requires fork should explicitly specify that via get_context() or set_start_method()

Contexts and start methods (cont'd)

仅供了解

- forkserver
  - When the program starts and selects the forkserver start method, a server process is spawned. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded unless system libraries or preloaded imports spawn threads as a side-effect so it is generally safe for it to use os.fork(). No unnecessary resources are inherited
  - Available on POSIX platforms which support passing file descriptors over Unix pipes such as Linux
  - Changed in version 3.8: On macOS, the spawn start method is now the default. The fork start method should be considered unsafe as it can lead to crashes of the subprocess as macOS system libraries may start threads. See bpo-33725
  - To select a start method you use the set_start_method() in the if __name__ == '__main__' clause of the main module.
  - set_start_method() should not be used more than once in the program.
  - Alternatively, you can use get_context() to obtain a context object. Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program.

spawn() VS. fork()

仅供了解

- Differences between spawn and fork
  - fork is fast, unsafe, and maybe bloated.
    - fork follows the COW (Copy-on-Write) rule. Requesting the process means calling the pointer. Modify the process will start to copy a private copy to the caller.
    - But! fork won't inherit threadings from parents, which may lead to deadlock.
  - spawn is safe, compact, and slower
    - Spawn starts a Python child process from scratch without the parent process's memory, file descriptors, threads, etc
    - Technically, spawn forks a duplicate of the current process, then the child immediately calls exec to replace itself with a fresh Python, then asks Python to load the target module and run the target callable
    - Slow because Python has to load, initialize itself, read files, load and initialize modules, etc

Process and exceptions

`class multiprocessing.Process(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)`

- The multiprocessing package mostly replicates the API of the threading module.
- Process objects represent activity that is run in a separate process. The Process class has equivalents of all the methods of threading.Thread.
- In addition to the threading.Thread API, Process objects also support the following attributes and methods:
- pid
  - Return the process ID. Before the process is spawned, this will be None.
- exitcode
  - The child's exit code. This will be None if the process has not yet terminated.
- authkey
  - The process's authentication key (a byte string).

Process and exceptions (cont'd)

- sentinel
  - A numeric handle of a system object which will become "ready" when the process ends.
- terminate()
  - Terminate the process. On POSIX this is done using the SIGTERM signal; on Windows TerminateProcess() is used. Note that exit handlers and finally clauses, etc., will not be executed.
  - Note that descendant processes of the process will not be terminated – they will simply become orphaned.
- kill()
  - Same as terminate() but using the SIGKILL signal on POSIX.
- close()
  - Close the Process object, releasing all resources associated with it. ValueError is raised if the underlying process is still running. Once close() returns successfully, most other methods and attributes of the Process object will raise ValueError.

Process Example

```python
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

```
main line
module name: __main__
parent process: 46095
process id: 46098
function f
module name: __mp_main__
parent process: 46098
process id: 46100
hello bob
```

Concurrent.futures (仅供了解)

Pool

- concurrent.futures.ProcessPoolExecutor offers a higher level interface to push tasks to a background process without blocking execution of the calling process. Compared to using the Pool interface directly, the concurrent.futures API more readily allows the submission of work to the underlying process pool to be separated from waiting for the results.
- Why we need Pool: taxi company, we don’t need to buy a car and hire a driver every time

concurrent.futures — Launching parallel tasks

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
- 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.
- 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.
- shutdown(wait=True, *, cancel_futures=False)
  - 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.

https://docs.python.org/3/library/concurrent.futures.html
https://peps.python.org/pep-3148/

Future Object

The Future class encapsulates the asynchronous execution of a callable.

- 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 or finished running 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 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 an exception, this method will raise the same exception.

Module Functions: wait()

- 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. Duplicate futures given to fs are removed and will be returned only once. Returns a named 2-tuple of sets. The first set, named done, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named not_done, contains the futures that did not complete (pending or running 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:
  - 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.

Module Functions: as_completed()

- 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 cancelled futures). 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 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.

- executor.map(), which executes tasks concurrently and returns results in the order they were submitted
- executor.submit() along with concurrent.futures.as_completed(), which also executes tasks concurrently but allows you to process results as they become available, regardless of the order of submission.
- However, we also store the returned Future objects in a dictionary called futures and use concurrent.futures.as_completed() to process the results as they become available, regardless of the order in which they were submitted.

https://www.packetswitch.co.uk/what-is-concurrent-futures-and-how-can-it-boost-your-python-performance/

ThreadPoolExecutor

- ThreadPoolExecutor is an Executor subclass that uses a pool of threads to execute calls asynchronously.
- class concurrent.futures.ThreadPoolExecutor(max_workers=None, thread_name_prefix='', initializer=None, initargs=())

ProcessPoolExecutor

- class concurrent.futures.ProcessPoolExecutor(max_workers=None, mp_context=None, initializer=None, initargs=(), max_tasks_per_child=None)

Async IO

Challenge from Practice

聂卫平车轮战12位小棋手。限时：聂卫平5秒，小棋手60秒

中国茶道：烧水、洗茶具、泡茶

- 下棋
  - 当前小棋手思考的时候，聂卫平等待这位小棋手思考，下子；然后，聂卫平和下一位小棋手对战
  - 当前小棋手思考的时候，聂卫平可以和下一位小棋手对战
- 泡茶
  - 烧水的时候，我们等待水烧开后，再去洗茶具
  - 烧水的时候我们可以同时洗茶具
- 回到程序中，如果把聂卫平或者我们看做CPU，应该怎么做，提高效率
- 单CPU单线程中，某些代码不需要等待前面的执行结果，可以将整个代码分割，异步(Async)执行

Concurrency and Parallelism

- Parallelism consists of performing multiple operations at the same time. Multiprocessing is a means to effect parallelism, and it entails spreading tasks over a computer's central processing units (CPUs, or cores). Multiprocessing is well-suited for CPU-bound tasks: tightly bound for loops and mathematical computations usually fall into this category.
- Concurrency is a slightly broader term than parallelism. It suggests that multiple tasks have the ability to run in an overlapping manner. (There's a saying that concurrency does not imply parallelism.)
- Threading is a concurrent execution model whereby multiple threads take turns executing tasks. One process can contain multiple threads. Python has a complicated relationship with threading thanks to its GIL, but that's beyond the scope of this article.
- What's important to know about threading is that it's better for IO-bound tasks. While a CPU-bound task is characterized by the computer's cores continually working hard from start to finish, an IO-bound job is dominated by a lot of waiting on input/output to complete.
- To recap the above, concurrency encompasses both multiprocessing (ideal for CPU-bound tasks) and threading (suited for IO-bound tasks). Multiprocessing is a form of parallelism, with parallelism being a specific type (subset) of concurrency. The Python standard library has offered longstanding support for both of these through its multiprocessing, threading, and concurrent.futures packages.

Async IO (3.7+)

asyncio: async, await. Async IO is a single-threaded, single-process design

- Async IO is a concurrent programming design that has received dedicated support in Python, evolving rapidly from Python 3.4 through 3.7, and probably beyond. (必须用3. 7或更新版本)
- Asynchronous IO (async IO): a language-agnostic paradigm (model) that has implementations across a host of programming languages
  - async/await: two new Python keywords that are used to define coroutines
  - asyncio: the Python package that provides a foundation and API for running and managing coroutines
- Now it’s time to bring a new member to the mix.

Over the last few years, a separate design has been more comprehensively built into CPython: asynchronous IO, enabled through the standard library’s asyncio package and the new async and await language keywords. To be clear, async IO is not a newly invented concept, and it has existed or is being built into other languages and runtime environments, such as Go, C#, or Scala.

- The asyncio package is billed by the Python documentation as a library to write concurrent code. However, async IO is not threading, nor is it multiprocessing. It is not built on top of either of these.

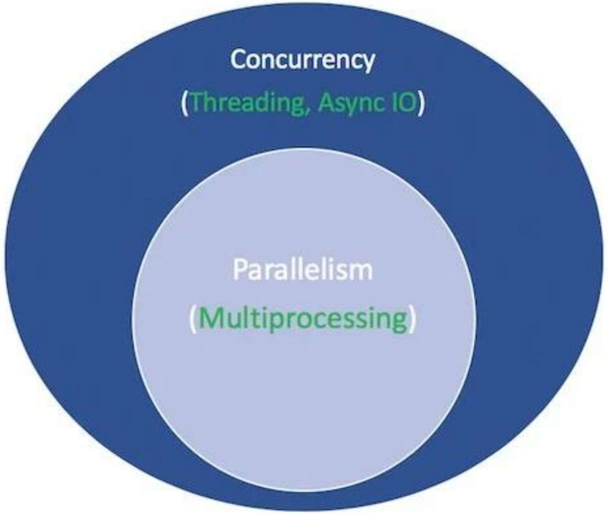
https://realpython.com/async-io-python/

Async IO (3.7+) (cont'd)

- In fact, async IO is a single-threaded, single-process design: it uses cooperative multitasking, a term that you'll flesh out by the end of this tutorial. It has been said in other words that async IO gives a feeling of concurrency despite using a single thread in a single process. Coroutines (a central feature of async IO) can be scheduled concurrently, but they are not inherently concurrent.
- To reiterate, async IO is a style of concurrent programming, but it is not parallelism. It's more closely aligned with threading than with multiprocessing but is very much distinct from both of these and is a standalone member in concurrency's bag of tricks.
- That leaves one more term. What does it mean for something to be asynchronous? This isn't a rigorous definition, but for our purposes here, I can think of two properties:

Async IO (3.7+) (cont’d)

- Asynchronous routines are able to “pause” while waiting on their ultimate result and let other routines run in the meantime.
- Asynchronous code, through the mechanism above, facilitates concurrent execution. To put it differently, asynchronous code gives the look and feel of concurrency.
- Here’s a diagram to put it all together. The white terms represent concepts, and the green terms represent ways in which they are implemented or effected:



asyncio VS. Threading

“Use async IO when you can; use threading when you must.”

- I’ve heard it said, “Use async IO when you can; use threading when you must.” The truth is that building durable multithreaded code can be hard and error-prone. Async IO avoids some of the potential speedbumps that you might otherwise encounter with a threaded design.
- But that’s not to say that async IO in Python is easy. Be warned: when you venture a bit below the surface level, async programming can be difficult too! Python’s async model is built around concepts such as callbacks, events, transports, protocols, and futures—just the terminology can be intimidating. The fact that its API has been changing continually makes it no easier.
- Luckily, asyncio has matured to a point where most of its features are no longer provisional, while its documentation has received a huge overhaul and some quality resources on the subject are starting to emerge as well.
- Python’s asyncio package (introduced in Python 3.4) and its two keywords, async and await, serve different purposes but come together to help you declare, build, execute, and manage asynchronous code.

Caution

- A Word of Caution: Be careful what you read out there on the Internet. Python's async IO API has evolved rapidly from Python 3.4 to Python 3.7. Some old patterns are no longer used, and some things that were at first disallowed are now allowed through new introductions.

Async IO: Example

Async cannot run under Jupyter

```python
import asyncio


async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```python
import time

def count():
    print("One")
    time.sleep(1)
    print("Two")

def main():
    for _ in range(3):
        count()

if __name__ == "__main__":
    s = time.perf_counter()
    main()
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

Async IO: Example (cont'd)

- While using time.sleep() and asyncio.sleep() may seem banal, they are used as stand-ins for any time-intensive processes that involve wait time. (The most mundane thing you can wait on is a sleep() call that does basically nothing.) That is, time.sleep() can represent any time-consuming blocking function call, while asyncio.sleep() is used to stand in for a non-blocking call (but one that also takes some time to complete).
- As you'll see in the next section, the benefit of awaiting something, including asyncio.sleep(), is that the surrounding function can temporarily cede control to another function that's more readily able to do something immediately. In contrast, time.sleep() or any other blocking call is incompatible with asynchronous Python code, because it will stop everything in its tracks for the duration of the sleep time.

async

- The syntax async def introduces either a native coroutine or an asynchronous generator. The expressions async with and async for are also valid, and you'll see them later on.
- The keyword await passes function control back to the event loop. (It suspends the execution of the surrounding coroutine.) If Python encounters an await f() expression in the scope of g(), this is how await tells the event loop, "Suspend execution of g() until whatever I'm waiting on—the result of f()—is returned. In the meantime, go let something else run."
- In code, that second bullet point looks roughly like this:
- async def g():

```
    # Pause here and come back to g() when f() is ready
    r = await f()
    return r
```

async/await

- A function that you introduce with async def is a coroutine. It may use await, return, or yield, but all of these are optional. Declaring async def noop(): pass is valid:
    - Using await and/or return creates a coroutine function. To call a coroutine function, you must await it to get its results.
    - It is less common (and only recently legal in Python) to use yield in an async def block. This creates an asynchronous generator, which you iterate over with async for. Forget about async generators for the time being and focus on getting down the syntax for coroutine functions, which use await and/or return.
    - Anything defined with async def may not use yield from, which will raise a SyntaxError.
- Just like it’s a SyntaxError to use yield outside of a def function, it is a SyntaxError to use await outside of an async def coroutine. You can only use await in the body of coroutines.

```python
async def f(x):
    y = await z(x)  # OK - `await` and `return` allowed in coroutines
    return y

async def g(x):
    yield x  # OK - this is an async generator

async def m(x):
    yield from gen(x)  # No - SyntaxError

def m(x):
    y = await z(x)  # Still no - SyntaxError (no `async def` here)
    return y
```

await

- Finally, when you use await f(), it’s required that f() be an object that is awaitable. Well, that’s not very helpful, is it? For now, just know that an awaitable object is either (1) another coroutine or (2) an object defining an .__await__() dunder method that returns an iterator. If you’re writing a program, for the large majority of purposes, you should only need to worry about case #1.
- That brings us to one more technical distinction that you may see pop up: an older way of marking a function as a coroutine is to decorate a normal def function with @asyncio.coroutine. The result is a generator-based coroutine. This construction has been outdated since the async/await syntax was put in place in Python 3.5.
- These two coroutines are essentially equivalent (both are awaitable), but the first is generator-based, while the second is a native coroutine:

asyncio.coroutine (outdated)

```python
import asyncio

@asyncio.coroutine
def py34_coro():
    """Generator-based coroutine, older syntax"""
    yield from stuff()

async def py35_coro():
    """Native coroutine, modern syntax"""
    await stuff()
```

- If you're writing any code yourself, prefer native coroutines for the sake of being explicit rather than implicit. Generator-based coroutines will be removed in Python 3.10.
- Towards the latter half of this tutorial, we'll touch on generator-based coroutines for explanation's sake only. The reason that async/await were introduced is to make coroutines a standalone feature of Python that can be easily differentiated from a normal generator function, thus reducing ambiguity.
- Don't get bogged down in generator-based coroutines, which have been deliberately outdated by async/await. They have their own small set of rules (for instance, await cannot be used in a generator-based coroutine) that are largely irrelevant if you stick to the async/await syntax.

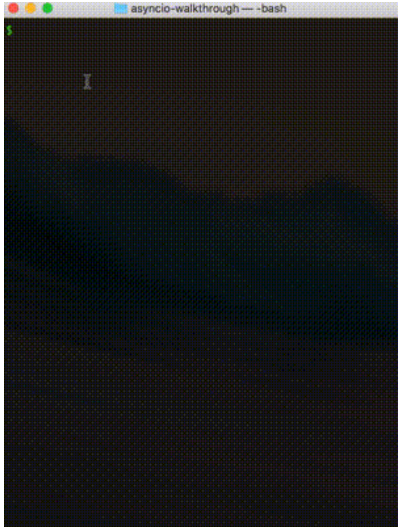
```python
import asyncio
import random

# ANSI colors
c = (
    "\033[0m",   # End of color
    "\033[36m",  # Cyan
    "\033[91m",  # Red
    "\033[35m",  # Magenta
)

async def makerandom(idx: int, threshold: int = 6) -> int:
    print(c[idx + 1] + f"Initiated makerandom({idx}).")
    i = random.randint(0, 10)
    while i <= threshold:
        print(c[idx + 1] + f"makerandom({idx}) == {i} too low; retrying.")
        await asyncio.sleep(idx + 1)
        i = random.randint(0, 10)
    print(c[idx + 1] + f"---> Finished: makerandom({idx}) == {i}" + c[0])
    return i

async def main():
    res = await asyncio.gather(*(makerandom(i, 10 - i - 1) for i in range(3)))
    return res

if __name__ == "__main__":
    random.seed(444)
    r1, r2, r3 = asyncio.run(main())
    print()
    print(f"r1: {r1}, r2: {r2}, r3: {r3}")
```



Summary

- 从CPU的角度考虑问题
- 充分利用多CPU并行能力
- 利用问题的并行结构
- 使用中，协程方案 优先于 线程方案
- 多线程是未来计算的基本范式
- 考虑到GIL，Python的多线程在3.12后可能迎来大的变革，所以目前掌握基本即可

Regular Expression (选修)

Regular Expression

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module.

- Find all the substring “a?b” in a text, like “aabbccaxbadbaab”, where “?” denotes an arbitrary character
- Find all the email addresses in a webpage? xxxx@sjtu.edu.cn
- In 1951, mathematician Stephen Cole Kleene described the concept of a regular language, a language that is recognizable by a finite automaton and formally expressible using regular expressions.
- In the mid-1960s, computer science pioneer Ken Thompson, one of the original designers of Unix, implemented pattern matching in the QED text editor using Kleene’s notation.
- Since then, regexes have appeared in many programming languages, editors, and other tools as a means of determining whether a string matches a specified pattern. Python, Java, and Perl all support regex functionality, as do most Unix tools and many text editors.

Kenneth Lane Thompson
(1943. Turing Award 1983)
Unix OS
B, Go Programming Lang.
Regular expressions
QED, ed text editor
UTF-8 encoding

RE

Metacharacters: . ^ $ * + ? { } [ ] \ | ( )

- Special metacharacters don’t match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched
    - Wildcare _ in structure pattern matching
    - Escape character in str
- []: They’re used for specifying a character class, which is a set of characters that you wish to match
- Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-’.
    - [abc] will match any of the characters a, b, or c; this is the same as [a-c], which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be [a-z].
- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions

```python
import re

p = re.compile('[b-d]')
print(p, type(p))

m = p.match("1a2b3c4d5e6f7z")
print(m, type(m))

m = p.search("1a2b3c4d5e6f7z")
print(m, type(m))

m = p.findall("1a2b3c4d5e6f7z")
print(m, type(m))

m = p.finditer("1a2b3c4d5e6f7z")
print(m, type(m))
```

```
re.compile('[b-d]') <class 're.Pattern'>
None <class 'NoneType'>
<re.Match object; span=(3, 4), match='b'> <class 're.Match'>
['b', 'c', 'd'] <class 'list'>
<callable_iterator object at 0x0000013F738CB550> <class 'callable_iterator'>
```

Matching method

```python
import re

p = re.compile('[b-d]')
print(p, type(p))

txt = "cs1a2b3c4d5e6f7z"

m = p.match(txt)
print(m, type(m))

m = p.search(txt)
print(m, type(m))

m = p.findall(txt)
print(m, type(m))

m = p.finditer(txt)
print(m, type(m))
for _ in m:
    print(_)
```

Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string
`search()`	Scan through a string, looking for any location where this RE matches
`findall()`	Find all substrings where the RE matches, and returns them as a list
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator

- match(): 从开头匹配。search()：任意匹配
- findall(): 所有匹配，保存在list。finditer(): 所有匹配，表示为迭代器
- Match object: span, match

```
re.compile('[b-d]') <class 're.Pattern'>
<re.Match object; span=(0, 1), match='c'> <class 're.Match'>
<re.Match object; span=(0, 1), match='c'> <class 're.Match'>
['c', 'b', 'c', 'd'] <class 'list'>
<callable_iterator object at 0x000002B46C39B5E0> <class 'callable_iterator'>
<re.Match object; span=(0, 1), match='c'>
<re.Match object; span=(5, 6), match='b'>
<re.Match object; span=(7, 8), match='c'>
<re.Match object; span=(9, 10), match='d'>
```

Matching characters (1)

- Metacharacters (except \) are not active inside classes.
  - For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; '$' is usually a metacharacter, but inside a character class it's stripped of its special nature.
- You can match the characters not listed within the class by complementing the set. This is indicated by including a '^' as the first character of the class.
  - For example, [^5] will match any character except '5'. If the caret appears elsewhere in a character class, it does not have special meaning. For example: [5^] will match either a '5' or a '^'.
- Perhaps the most important metacharacter is the backslash, \. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It's also used to escape all the metacharacters so you can still match them in patterns; for example, if you need to match a [or \, you can precede them with a backslash to remove their special meaning: \[or \\.
- Some of the special sequences beginning with '\' represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn't whitespace.

Matching characters (2)

- \d Matches any decimal digit; this is equivalent to the class [0-9].
- \D Matches any non-digit character; this is equivalent to the class [^0-9].
- \s Matches any whitespace character; this is equivalent to the class [\t\n\r\f\v].
- \S Matches any non-whitespace character; this is equivalent to the class [^ \t\n\r\f\v].
- \w Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_].
- \W Matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9_].
- These sequences can be included inside a character class. For example, [\s,.] is a character class that will match any whitespace character, or ',' or '.'.
- The final metacharacter in this section is .. It matches anything except a newline character, and there's an alternate mode (re.DOTALL) where it will match even a newline. . is often used where you want to match "any character".

https://docs.python.org/3/library/re.html#re-syntax

Repeating things (1)

- The first metacharacter for repeating things that we'll look at is *. * doesn't match the literal character '*'; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.
- For example, ca*t will match 'ct' (0 'a' characters), 'cat' (1 'a'), 'caaat' (3 'a' characters), and so forth.
- Repetitions such as * are greedy; when repeating a RE, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don't match, the matching engine will then back up and try again with fewer repetitions.
- Another repeating metacharacter is +, which matches one or more times. Pay careful attention to the difference between * and +; * matches zero or more times, so whatever's being repeated may not be present at all, while + requires at least one occurrence. To use a similar example, ca+t will match 'cat' (1 'a'), 'caaat' (3 'a's), but won't match 'ct'.

Repeating things (2)

- There are two more repeating operators or quantifiers. The question mark character, ?, matches either once or zero times; you can think of it as marking something as being optional. For example, home-?brew matches either 'homebrew' or 'home-brew'.
- The most complicated quantifier is {m,n}, where m and n are decimal integers. This quantifier means there must be at least m repetitions, and at most n. For example, a/{1,3}b will match 'a/b', 'a//b', and 'a///b'. It won't match 'ab', which has no slashes, or 'a////b', which has four.
- You can omit either m or n; in that case, a reasonable value is assumed for the missing value. Omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity.
- Readers of a reductionist bent may notice that the three other quantifiers can all be expressed using this notation. {0,} is the same as *, {1,} is equivalent to +, and {0,1} is the same as ?. It's better to use *, +, or ? when you can, simply because they're shorter and easier to read.

Using Regular Expressions

- The re module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them
- Regular expressions are compiled into pattern objects
- re.compile() also accepts an optional flags argument, used to enable various special features and syntax variations. We'll go over the available settings later, but for now a single example will do:

```
p = re.compile('ab*', re.IGNORECASE)
```

- The RE is passed to re.compile() as a string. REs are handled as strings because regular expressions aren't part of the core Python language, and no special syntax was created for expressing them. (There are applications that don't need REs at all, so there's no need to bloat the language specification by including them.) Instead, the re module is simply a C extension module included with Python, just like the socket or zlib modules.

Matching object

Method/Attribute	Purpose
group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

```python
import re
p = re.compile('[a-z]+')

m = p.match('tempo')

print(m.group())
print(m.start(), m.end())
print(m.span())
```

```
tempo
0 5
(0, 5)
```

```python
import re
p = re.compile('[a-z]+')
print(p.match('::: message'))
m = p.search('::: message')
print(m)
print(m.group())
print(m.span())
```

```
None
<re.Match object; span=(4, 11), match='message'>
message
(4, 11)
```

Module-Level Functions

```python
import re

print(re.match(r'From\s+', 'Fromage amk'))
print(re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998'))

print(re.match('[a-z]+', '::: message'))
print(re.search('[a-z]+', '::: message'))
```

```
None
<re.Match object; span=(0, 5), match='From '>
None
<re.Match object; span=(4, 11), match='message'>
```

- You don’t have to create a pattern object and call its methods; the re module also provides top-level functions called match(), search(), findall(), sub(), and so forth. These functions take the same arguments as the corresponding pattern method with the RE string added as the first argument, and still return either None or a match object instance
- If you use a particular regex in your Python code frequently, then precompiling allows you to separate out the regex definition from its uses. This enhances modularity
- In theory, you might expect precompilation to result in faster execution time as well. In practice, though, that isn’t the case. The truth is that the re module compiles and caches a regex when it’s used in a function call. If the same regex is used subsequently in the same Python code, then it isn’t recompiled. The compiled value is fetched from cache instead. So the performance advantage is minimal

Reference

- https://en.wikipedia.org/wiki/Ken_Thompson
- https://docs.python.org/3/howto/regex.html
- https://realpython.com/regex-python/
- https://realpython.com/regex-python-part-2/
- grep command in Ubuntu
- “Mastering Python Regular Expressions”, by Felix Lopez, Victor Romero
- “Mastering Regular Expressions”, 3rd Edition, by Jeffrey Friedl (Author)