The Python Language

Variable: Letters, numbers, or underscores; CaSe sEnSiTiVe; Not allowed: starting with number; **Reserved words**: and, del, for, is, raise, assert, elif, from, lambda, return, break, else, global, not, try, class, except, if, or, while, continue, exec, import, pass, yield, def, finally, in, print

Datatype: Integer: 1,2,3 (int, Unbounded); Float: 0.1, 0.4, 4.0 (float, Digits and Exponents); String (str, Series of Unicode characters, Character: String of length 1, Enclosed by a pair of single or double quotes, Multiline: triple quote (Double or single)); Boolean: True, False; List: [1,2,3]; Tuples: (1,2,3)

Operation String: Substring: string[index:end:step]; + concatenation; * multiple concatenation; in, not in contains/not contains; Format print(f"Greeting, {name}. You are {age}")

Indentation Rule: Increase indent after an if statement or for statement (after :) (Equivalent to C, Java's {); Maintain indent to indicate the scope of the block (Which lines are affected by the if/for); Reduce indent to back to the level of the if statement or for statement to indicate the end of the block (Equivalent to C, Java's }); Blank lines are ignored - they do not affect indentation; Comments on a line by themselves are ignored with reference to indentation; Python cares a lot about how far line is indented; Don't mix tabs and spaces; Use one only (Most text editors can turn tabs into spaces - make sure to enable this feature)

Function: Group of related statements performing a specific task; Break programs into small chunks; Better code organization; Code reusable

Definition: Function Name; Parentheses; Arguments

Collection

Sets*: Unordered collection of items; No duplication; Operators: s1.isdisjoint(s2): no common element; s1 <= s2, s1.issubset(s2): s1 ⊆ s2; s1 >= s2, s1.issuperset(s2): s1 ⊇ s2; s3 = s1 | s2, s3 = s1.union(s2): s3 = s1 ∪ s2; s3 = s1 & s2, s3 = s1.intersection(s2): s3 = s1 ∩ s2; s3 = s1 - s2, s3 = s1.difference(s2): s3 = s1 s2*

Sequences: *Ordered collection of items; Can have duplications; Positioned access; Slicing similar to strings - seq[start:end:step]; Implementations (list, tuple, range); str*

Lists: *Mutable sequence (Values can be changed late); Flexible, widely used; Comma separated declaration; + append elements at the end, same or .extend(); = replaces single value or bunch of values; sort() elements; del delete elements like del a[i]; .remove() occurrences like a.remove('a')*

Tuples: *Immutable sequence; Contain any type of element.; A very common use of tuples is a simple representation of pairs like Positition (x, y), size(w,h)*

Map: *Key/value pairs (Key must be unique, Similar to JSON objects); Unordered, mutable;Implemented by dict;Key operations: in, not in: check key presence; max, min of key; Operation: d[k]: get value by key; d[k] = v: set value to key; del d[k] remove key from dict; Method:*

d.get(k[, default]): same as d[k], fallback to default if key not found; **d.pop(k[, default]):** del d[k] and return previously; **deleted d[k]**, fallback to default if key not found; **d1.update(d2)**: for each key in d2, sets d1[key] to d2[key], replacing the existing value if there was one; **d.keys()**: returns list of keys; **d.values()**: returns list of values; **d.items()**: returns list of (key,value) tuples.

Loop: Loops (repeated steps) have iteration variables; Iteration variable changes each time through a loop; Often these iteration variables go through a sequence of numbers; keyword: break to exit loop early; continue to skip current iteration; pass to do nothing; range to iterate over a sequence of numbers

OOP in Python

Procedural Programming: Variables and related functions are separated; Programs is divided into functions

Object-Oriented Programming: Variables and related functions are bound together; Programs are divided into objects

Some method: __init__, __str__, __repr__, __del__, __lt__(self, other) (self.a < other.a)

Inheritance (Tính kế thừa): Define a child class with a superclass in parentheses; All methods and attributes from superclass will be inherited to subclass

```
class Person:
class President(Person):
class President(Person, Employee):
macron = President("Emmanuel Macron", 43)
```

Checking inheritance: Built-in functions isinstance() and issubclass(): **isinstance()** returns True if the object is an instance of the class or other classes derived from it; **issubclass()** is used to check for class inheritance.

Polymorphism (Tính đa hình): A superclass's method can be overridden, simply by deffing the same method name in the subclass; A superclass instance can be accessed using super() in the subclass

```
class Person:
class President(Person, Employee):
def describe(self):
    super().describe()
    print("Term:", self.term)
def work(self):
    super().work() # from Employee
```

Encapulation (Tính đóng gói): public by default; No specified keyword; Use underscore prefixes(name: public; _name: protected; __name: private); Accessor methods / Mutator methods; Getter / Setter

Modules and packages: Modularity; Reusability; Shareibility; Maintainability

Package: A bunch of related modules; Why(Higher level of modularity; Less import modules from the same packages; Module name A.B designates a submodule named B in a package named A.)

Files and Directories

Files: Everything in UNIX is a file; File name is the name of the file; Why(RAM is volatile;File is persistent)

Open file: open(fileName, mode) - fileName: what file; mode: what operations; returns a File object representing an opened file

Mode(r reading, default; w Writing. Creates or clears a file; x Exclusive creation. Fails if file exists; a Appending. Creates if file does not exist.; t Opens in text mode. (default); b Opens in binary mode.; `+ Opens a file for updating (rw))

Write and Read file

- **f.read(size)** reads and returns size bytes; size is optional; Reads all file content by default; Updates current file pointer after .read()Be careful for large files!
- **f.seek(offset)** sets current file pointer to a specific offset
- **f.write()** writes into file; Text files: .readline(): reads until a new line. (There's a at the end of file); .readlines(): reads all lines

Buffering: in-memory cache of file content: Speeding up IO accesses; Reading/writing blocks is faster than individual bytes

Use open(fileName, mode, buffering = -1): buffering is optional (-1, same as io.DEFAULT_BUFFER_SIZE; 0: disable buffering; 1: line buffering for text files; >1: fixed size buffer); Flushing buffer: write buffer to disk, if any (Manually f.flush())

Close file: *Close a file after using; Clean up OS caches, buffers; Without closing, there may be data loss with power outage (f.close())*

Extra

Exceptions: Python: try… except…; For handling IO errors (IOError)

Temporary files: Don't care about name, location; Just somewhere to store temp contents; Automatically cleaned up after close()

```
import tempfile.TemporaryFile
# gimme a file, whenever it is
f = tempfile.TemporaryFile('w+t')
f.write("3.1415926...")
f.close()
```

Compression: What: Use less storage to represent data;Why(Smaller disk storage; Easier for transmission over network; Encryption with passwords); Plenty of existing modules (zlib, gzip, bz2, lzma, tarfile, zipfile); Each module would have different advantages/disadvantages and usage

Objects: Serialize objects into byte array(Save state to disk, optionally compressed (!); Load state later; Transmit object to a remote machine)

Directions: Hierachical structure(A bunch of files; A bunch of sub-directories); Looks like a tree(Path indicates a location inside a directory); Why(For organization of data; Easier traversing and browsing); How(Listing: os.scandir(), os.walk() (recursive); Creating: os.mkdir(), os.mkdirs()Deleting: os.rmdir(), shutil.rmtree() (recursive))

Multi Processing

Process: Process is a program in execution state (active); Why process? (Program is passive; No execution → what's running?); A process execution state contains(Processor state (context); File descriptors; Memory allocation(Process stack; Data section; Heap)

Process States: New (admitted) -> ready (<-interrupt) (-> scheduled) running running -> (event wait) waiting waiting -> (event finish) ready runing -> (exit) terminated

new: process has just been created; ready: waiting to be assigned (scheduled) to a processor; running: it's executing instructions; waiting: waiting for some events to occur; terminated: finished execution

Process Creation: Start a new process == Create a new process; Create new child process (Can create child process → grand child process); Dependent on OS, parent and child can share(All resources: opened files, devices, etc. . .; Some resources: opened files only; No resource); ; A fully loaded system will have a process tree; Unix (fork() - parents >0; child 0) and exec()

Scheduling: Multiple processes running at the same time; Process scheduler is a part that decides which processes to be executed at a certain time.; Maximize CPU usage; Responsiveness for User interface; Provide computational power for heavy-workload processes; «Multitasking»; **Different characteristics of processes**(CPU bound: spends more time on computation; I**/O bound: spends more time on I/O devices** (reading/writing disk, printing. . .)); **By the ability to pause running processes(**Preemption: OS forcely pauses running processes; Non-preemption (also cooperation): processes willing to pause itself); **By duration between each «switch»**(Short term scheduler: milliseconds (fast, responsive); Long term scheduler: seconds/minutes (batch jobs))

Scheduling with Context Switch: Switch between processes(Save data of old process; Load previously saved data of new process); **Context switch is overhead**(No work done for processes during context switch; Time slice (time between each switch) is hardware-limited)

Scheduler: Knowns(List of processes; Process states; Accounting information); **Constraints (Process priority (if any))**(Processes have scheduling priority; Indicates the importance of each process; Higher priority: more likely to be scheduled);

Problem 1: What processes to run next? Job queue - set of all processes entering the system, storedon disk; Ready queue - set of all processes residing in main memory, ready and waiting to execute; Device queues - set of processes waiting for an I/O device; Lists of PCBs; Processes change state → they migrate among the various queues

Problem 2: How long should it run? First In First Served; Earliest Deadline First; Shortest Remaining Time; Round Robin

Algorithm	Preempt?	Priority?	Note
First Come, First Served	No	No	Depends on arrival time
Shortest-Job-First	No	Yes	Low waiting time ω
Shortest-Remaining-Time-First	Yes	Yes	Preemptive SJF, low ω
Round Robin	Yes	No	Low response time ρ
Multilevel Queue	Depends	Depends	Several subqueues, permanent
Multilevel Feedback Queue	Depends	Depends	Several subqueues, migrate

Task

		Os module	Subprocess module
Create a process	Run and wait	• os.system("ps aux")	subprocess.run(["ps", "aux"])
	Run in background	os.system("long_command.sh &")	subprocess.Popen("long_command.sh")
	Timeout	N/A	subprocess.run("long_command.sh", timeout = 10
IO redirection	Redirect input	os.popen("bc", "w").write("1+2")	subprocess.Popen("bc", stdin=subprocess.PIPE).communicate(b"3+4")
	Redirect output	print(os.popen("ps aux", "r").readlines())	subprocess.Popen(["ps", "aux"], stdout=subprocess.PIPE).communicate()
	Redirect with pipe	• os.pipe(), os.fork()	subprocess.Popen("bc", stdin=anotherProcess.stdout)
	Terminate	• os.kill(pid, signal.SIGTERM)	anotherProcess.terminate(), anotherProcess.kill()
	Get return code	• return value of os.system()	Get return code subprocess.check_output(), catch CalledProcessError

Multithreading

Process Control Block contains: Process ID; Process state (new/ready/running/waiting/terminated); Processor state (program counter, registers); File descriptors; Scheduling information (next section); Accounting information (limits)

Thread & Single-threaded process

Thread: a single flow of execution; belongs to a process; can be considered as lightweight process

Single-threaded process: Default; Only one thread per process; (Single stack; Single text section (code); Single data section (global data); Single heap (dynamic allocation))

Multi-threaded process: More than one thread per process; Share the same PCB among threads (Process state; Memory allocation (heap, global data); File descriptors (files, sockets, etc.); Scheduling information; Accounting information); Different processor state (program counter, registers); Different stack

Multi-threaded process

Each thread has: Private stack; Private stack pointer; Private program counter; Private register values; Private scheduling policies

Share: Common text section (code); Common data section (global data); Common heap (dynamic allocation); File descriptors (opened files); Signals. . .

Multi-threaded process vs Multi process: Same goals(Do several things at the same time; Increase CPU utilization; Increase responsiveness); What is the principal difference between these two types of process? (Multi-process with fork(): «resource cloning»; Multi-thread process: «resource sharing»)

Responsiveness: Perform different tasks at the same time (Several operations can block (e.g. network, disk I/O); UI needs responsiveness) -> one thread for UI, other threads for background tasks

Performance: Creating (fork()) a new process is slower than a thread; Terminating a process is also slower than a thread; Switching between processes is slower than between threads

Resource Sharing: Memory is always shared(Heap; global data); All file descriptors are also shared(Open files; TCP sockets; UNIX sockets; Devices); No need to use shm*()

Scalability: More CPU cores: simply increase number of threads; Don't create too many threads (Overhead; Synchronization)

⇨ Why not multithread: Nondeterministic; Synchronization; Deadlock and Complication

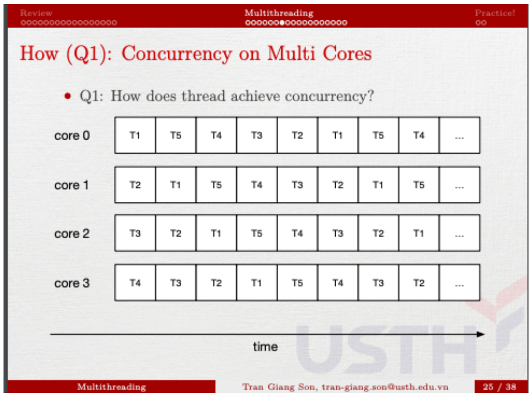
Multithreading

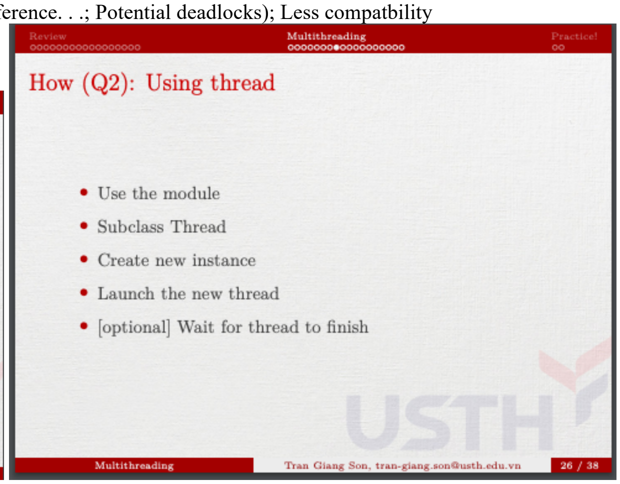
Global Interpreter Lock (GIL): Implemented in CPython; Mutex

Only 1 thread can control the Python intepreter; Only one thread can be executed at any given time (Bottleneck in Python CPU-bound code; Not a problem in wrapper-to-native-code; Not a problem in IO-bound programs)

Why GIL? Memory management (Reference counting; Garbage collector); Simplification of thread-safety (Only 1 mutex on the intepreter; No multiple mutexes on each object; No deadlock)

Removing GIL? Slower single-threaded performance (1 mutex per object reference. . .; Potential deadlocks); Less compatbility






```
import threading (Use the module)
import time
class BackgroundThread(threading.Thread): (Subclass Thread)
        def __init__(self, sleepTime): (passing parameter)
                threading.Thread.__init__(self)
                self.__sleepTime = sleepTime
        def run(self): (Override method to run in background)
                time.sleep(self.__sleepTime)
                print(f"Finished sleeping {self.__sleepTime}s")
backgroundThread = BackgroundThread(10) // Create new instance of the thread class
backgroundThread.start() // Launch the new thread with .start()
backgroundThread.join() // Waiting for finish with .join
print("Finished main thread")
```

GUI Toolkit