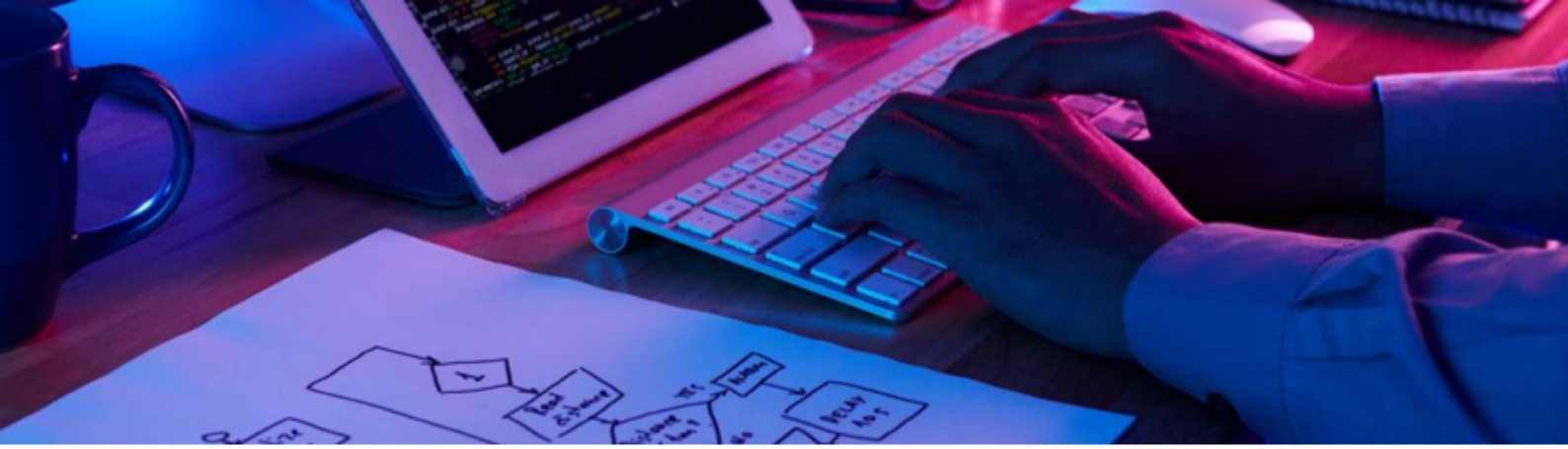


Troubleshooting and Debugging Techniques

Course 4 of 6 in Google IT Automation with Python Specialization.

bangkit

Troubleshooting and Debugging Techniques

Introduction to Debugging

- Whether it's an application crashing, a hardware issue, or network outage, IT specialists tend to run into the problem that needs to be solved regularly. When facing these issues, it's important to make sure that people affected by the problem can get back to doing their jobs as fast as possible.
- **Troubleshooting** is the process of identifying, analyzing, and solving problems. **Debugging** is the process of identifying, analyzing, and removing bugs in a system. Sometimes the terms used interchangeably. In general, troubleshooting is when fixing problem in the system running the application, and debugging is when fixing the bugs in the actual code of the application.

Introduction to Debugging

- The usual steps to solve almost any technical problem:
  1. Getting information
     - Includes information of reproduction case, which is a clear description of how and when the problem appears
  2. Finding the root cause of the problem
     - The key is to get to the bottom of what's going on, what triggered the problem, and how to change it.
  3. Performing the necessary remediation
     - This might include an immediate remediation to get the system back to health, and then a medium or long-term remediation to avoid the problem in the future.

Introduction to Debugging

- Sometimes an error program prints nothing on the screen. It simply just exits. Among the tools we can use to debug in Linux is **strace** command. It will trace a system calls made by the program and show the result of each of these calls. **System calls** are the calls that the programs running on the computer make to the running kernel.
- To understand more about the output of the strace and the related system calls, read more about each of them in the corresponding manual page, usually using **man** command.

Understanding the Problem

- When working with users, it's pretty common to receive reports of failures that just boil down to, "it doesn't work". But there are some common questions to users, such as:
  - What were you trying to do?
  - What steps did you follow?
  - What was the expected result?
  - What was the actual result?

Understanding the Problem

- Reproduction case is a way to verify if the problem is present or not. When trying to create a reproduction case, find the actions that reproduce the issue, and make these to be as simple as possible.
- How to figure out what's causing a problem?
  - The first step is to read the logs available
  - The next step is to try to isolate the conditions that trigger the issue

Understanding the Problem

- Once the reproduction case available, it might seem that it is the root cause of the problem. But more often than not, it's not true.
- Understanding the root cause is essential for performing the long-term remediation.
- To find out the actual root cause of the problem, it's generally by following a cycle of looking at the available information, coming up with a hypothesis that could explain the problem, and then testing the hypothesis. If the theory confirmed, then it's the root cause. If it isn't, then go back to the beginning and try different possibilities.

Understanding the Problem

- If the problem related with disk input and output, to get more information (in Linux), use **iotop** command. The purpose is to see which processes are using the most input and output. Other related tools are **iostat** and **vmstat** that show statistics on the input/output operations and the virtual memory operations. If the issue is the process itself that generates too much input or output, use **ionice** command to reduce its priority to access the disk.
- If the problem is related to the network traffic, check with **iftop** command that shows current traffic on network interfaces. To limit the bandwidth use a program like Trickle.
- If the problem is related to processing power, try the **nice** command to reduce the priority of the process when accessing the CPU.

Understanding the Problem

- For bugs occurring at random times, repair the system to give us as much information as possible when the bug happens.
- Sometimes, the bug goes away after adding extra logging information, or when following the code step by step using a debugger. This is an especially annoying type of intermittent issue, nicknamed “Heisenbug”. The term is coined in honor of Werner Heisenberg, the scientist who first described the **observer effect**, meaning a simple act of observing a phenomenon alters the phenomenon itself.
- Yet another type of intermittent issue is the one that goes away when we turn something off and on again. There’s almost certainly a bug in the software and the bug probably has something to do with failing to manage resource correctly.

Binary Searching a Problem

- When trying to find an element in a sorted list, use an efficient algorithm called binary search. In troubleshooting, this is applicable to go through and test a long list of hypotheses. When doing this, the list of elements contains all the possible causes of the problem and it keeps reducing the problem by half until only one option is left. This approach is sometimes called **bisecting** which means dividing in two.
- When using Git for version control, there's a Git command called **bisect** which receives two points in time in th Git history and repeatedly permits the user to try the code at the middle point between them until the commit that cause the breakage is found.

Understanding Slowness

- Monitor the usage of resources (CPU, memory, disk I/O, network, etc) to know which of them is being exhausted.
- In Linux systems, there are (command line) tools like **top**, **iotop**, and **iftop**. On MacOS, use an app called **Activity Monitor**. On Windows we will find tools called **Resource Monitor** and **Performance Monitor**.
- To diagnose what's causing the computer to run slow, the first step is to always open one of these tools. Check out what's going on and try to understand which resources lead to bottleneck and why. Then plan how to solve the issue.

Understanding Slowness

- To be fast, if you have a process that requires reading data repeatedly over the network. Afterwards, change the process to only read it once (from the network) & save the data into the disk. From thereon, such process will only be read only from the disk. Similarly, put information to the process memory to avoid loading it from disk every time. It's called the concept of **cache**.
- A **cache** stores data in a form that's faster to access than original form.
    - A web proxy is a form of cache that stores websites, images, or videos oftenly accessed by users so they don't need to download from the Internet every time.
    - The operating system also takes care of some caching. It keeps as much information as possible in RAM so fastly accessible. If there's still not enough RAM after that, the OS will put the parts of the memory that aren't currently in use onto the disk called **swap**.

Understanding Slowness

- If a program is using a lot of memory and this stops when the program restarted, it's probably because of a memory leak. A **memory leak** means that the memory that is no longer needed is not getting released.
- The ideal solution for a problem like this is to change the code so that it frees up some of the memory used. Without the code, another option is to schedule a regular restart to mitigate both the slow program and the computer running out of RAM.
- Whenever dealing with a computer that runs slowly, look at what the bottleneck is, figure out the root cause behind the resource being used up, and then take appropriate action.

Understanding Slowness

- Check web page load using a tool called **ab** which stands for Apache Benchmark.
- If the server is slow because of other process (Eg: ffmpeg), try to use **nice** or **renice** command to set lower priority of the process. Get process ID with **pidof** command.
- The other alternative is to send signal STOP to the concurrent processes, then send signal CONT to continue in serial. Use **kill**/**killall** command.

Slow Web Server

```
$ ab -n 500 site.example.com
…
Complete requests:   500
Requests per second: 6.43 [/s](mean)
Time per request:    155.4 [ms] (mean)
…

$ for pid in $(pidof ffmpeg); do
renice 19 $pid; done
```

19 is the lowest priority

```
$ killall -STOP ffmpeg
$ for pid in $(pidof ffmpeg); do
while kill -CONT $pid; do sleep 1;
done; done
```

continue the process one at a time

Slow Code

- A **profiler** is a tool that measures the resources that our code is using. It gives us a better understanding of what's going on. In particular, it helps to see how the memory is allocated and how the time spend.
- The profilers are specific to each programming language. For example, use gprof to analyze a C program, c-Profile module to analyze a Python program, and so on.
- To fix the slow code, one of the alternatives is to restructure it in order to avoid repeating **expensive actions** that take a long time to complete. Eg: parsing a file, reading data over the network or iterating through a whole list.

Slow Code

- Having a good understanding of the available data structures can help avoid unnecessary expensive operations and create efficient scripts.
- As a rule of thumb between **list** and **dictionary** in Python:
  - If the program needs to access elements by position or will always iterate through all the elements, use a list to store them.
  - If the program needs to look up the elements using a key, use dictionary.
  - But it doesn't make sense to create a dictionary and fill it with data if it is only going to lookup one value in it. Just iterate over the list and get the element.

Slow Code

- How if there's an expensive operation inside a loop? It multiplies time needed for expensive operation by the amount of time while repeating the loop.
- Whenever there’s a loop, make sure to check what actions in it, and see if there are operations to take out of the loop to do them just once.
- Make sure the list of elements that is iterated through is only as long as what it really needs to be. Another thing to remember about the loops it to break out of the loop once the program found what it's looking for.
- One last thing to keep in mind is that the right solution for one problem might not be right for other different problems.

Slow Code

- Another strategy to avoid doing expensive options is storing the intermediate results to avoid repeating the expensive operation more than needed. One of the common techniques is by creating a local **cache**.
- It is sometimes tricky to get the cache working properly. So think about how often is going to update the cache and what happens if the data in the case is out of date.
- Keep in mind that caches don't always need to be elaborate structures storing lots of information with a complex timeout logic. Sometimes, it can be as simple as having a variable that stores a temporary result instead of calculating this result every time.

Slow Code

- While calling a script/command, preceded with **time** command to prints how long the script took to execute it.
- **Real** is the amount of actual time that it took to execute the command (or called wall-clock time). **User** is the time spent doing operations in the user space. **Sys** it the time spent doing system level operations.

Slow Script with Expensive Loop

```
$ time ./send_reminders.py
"2020-12-12|Example|test1"
real    0m0.129s
user    0m0.068s
sys     0m0.013s

$ time ./send_reminders.py
"2020-12-12|Example|test1,test2,test3,
test4,test5,test6,test7,test8,test9"
real    0m0.296s
user    0m0.222s
sys     0m0.008s
```

Slow Code

- There's a bunch of different profilers available for Python that work for different use cases. This example uses **pprofile3** then exports the data to the **callgrind** format .
- The callgrind data (format) can be read by graphical interface based tools like **KCachegrind** or the other tools.

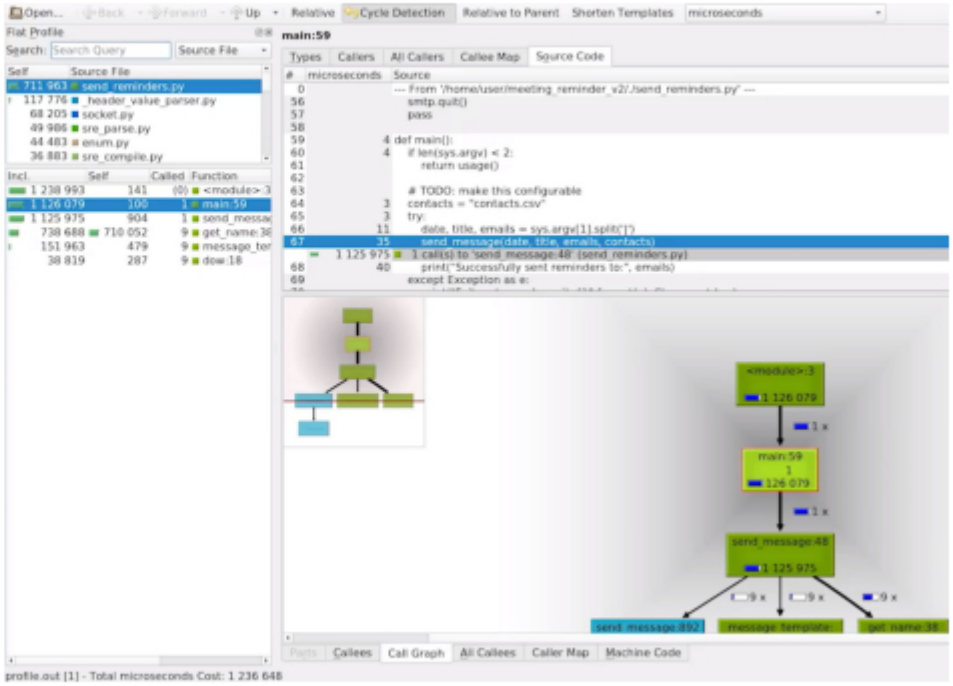
```
$ pprofile3 -f callgrind -o profile.out
./send_reminders.py
"2020-12-12|Example|test1,test2,test3,tes
t4,test5,test6,test7,test8,test9"
real    0m0.296s
user    0m0.222s
sys     0m0.008s

$ kcachegrind profile.out
# screenshot of the output on the next slide
```

bangbit

Slow Code

KCachegrind tools



When Slowness Problems Get Complex

- To be efficient, write a script to do operations in parallel. That way, while the computer is waiting for the slow I/O, other works can take place. The tricky part is dividing up the tasks so it gets the same result in the end.
- If there's a process that using a lot of CPU while a different process is using a lot of network I/O and another process is using a lot of disk I/O, they can all run in parallel without interfering with each other.
- There's actually a whole field of computer science called **concurrency**, dedicated on how to write programs that do operations in parallel.

When Slowness Problems Get Complex

- When using the processes in the OS to split the work, these **processes** don't share any memory. In the opposite, to share memory, use **threads**. Threads run parallel tasks inside a process, it share memory with other threads in the same process.
- In Python use the **Threading** or **AsyncIO** modules to implements threading. The implementation has to specify which parts of the code to run in separate threads or a separate asynchronous events, and how to combine the results of each in the end.

When Slowness Problems Get Complex

- Let's say there's a script to generate random pairs of people then send email. For small number of people it might just store the list of names and emails in a **CSV** file.
- At some point parsing the file will begin taking a whole lot of time, so consider to store the data in **SQLite** file. As such, lightweight database stored in the file without needing to run server.
- With more users coming in, keeping concurrency in a file would be hard, so it is time to use a **full-fledged database** server, probably even running on a separate machine.
- And there's even one more step after that. When the database isn't fast enough, add a caching service like **memcached** which keeps the most commonly used results in RAM.

When Slowness Problems Get Complex

- In Python **concurrent** module, to be able to run things in parallel, create an **executor**, the process in charge of distributing the work among the different workers.
- The **futures** submodule, (part of the **concurrent** module) provides a couple of different executors. One for using **threads** and another for using **process**.

Using Parallel Threads in Python

```python
from concurrent import futures

def main():
  # snipped some parts of code

  executor = futures.ThreadPoolExecutor()

  # call function with executor.submit()

  # wait all threads to finish
  executor.shutdown()
```

threads, parallel operations

When Slowness Problems Get Complex

- In previous code, the code running in threads. Now let's change the code to processes. In this (course's) case run faster than threads.
- Threads use a bunch of safety features to avoid having two threads that try to write to the same variable, which means end up waiting for their turn to write to variables for a few milliseconds.

Using Parallel Processes in Python

```
from concurrent import futures

def main():
  # snipped some parts of code

  executor = futures.ProcessPoolExecutor()

  # call function with executor.submit()

  # wait all threads to finish
  executor.shutdown()
```

processes, parallel operations

Why Programs Crash

- When a program terminates unexpectedly, go through usual cycle of gathering information about the crash, dig in until find the root cause, and then apply the right fix.
- To find the root cause of a crashing application look at all available logs, figure out what changed, trace the system or library calls the program makes, and create the smallest possible reproduction case.
- In order to reduce the scope of the problem, start with the actions that are easier and faster to check. To further reduce the scope, check if the error is just on the application or the whole system. Please also verify if the problem is related to the application (software) or hardware.

Why Programs Crash

- When an application crashes, look for logs that might relate to the failure. On Linux, open system log files in /var/log/ directory. On Mac OS generally use the Console app, and the Event Viewer on Windows.
- If there are no errors or the error logs aren't useful, try to find out more info by enabling debug logging. Eg: from a setting in the application configuration file or a command line parameter to pass when running the application manually.
- In that case you need to see what going on inside the program, what files & directories it's trying to open, what network connection it's trying to make, what information it's trying to read or write. On Linux use **strace** command to see what system calls a programs doing. Likewise, it is equal with **dtruss** on Mac OS or **process monitor** tool in Windows.

Why Programs Crash

- To debug an application that crashes, finding a reproduction case can help us both understand what's causing the crash and figure out what we can do to fix it.
- Spend some time figuring out the state that triggers the crash. It includes the overall system environment the specific application configuration, the inputs to the application, the outputs generated by the application, the resources that uses and the service it communicates with.

Why Programs Crash

- If you need to fix an application that crashes but can't change the code, figure out a way of working around the problem and avoiding the crash.
- For example, if the crashes only happen when the input isn't in the format the code expects, try to write a scripts that pre-process the data. If the problem is caused by an external service that the application uses and that's no longer compatible, try to write a **wrapper**, a function or program that provides a compatibility layer between two functions or programs, so they can work well together.
- Another possibility, is to match the environment with what the app recommended by developers. This could be running the same version of OS, using the same version of dynamic libraries, or using the same backend services.

Why Programs Crash

- If there's still no way to stop an application from crashing but if it crashes it starts back again, what to do? Try to deploy a watchdog, a process that checks whether a program is running and ,when it’s not, starts the program again.
- No matter how you work around the issue, remember to always report the bug to the application developers, include as much information as possible. If there’s a good reproduction case for the issue, it'll be easier for the developers to figure out what’s wrong and how to fix it.

Why Programs Crash

- Related to the web server with HTTP 500 error, or internal server error, check the log in the web server (Linux), located in /var/log/ directory.
- To get information about what program running in port 80, use **netstat** command, preceded with **sudo** command to get super user access, piped to **grep** command to filter only port 80 (:80).

```
sudo netstat -nlp | grep :80
```

- Enable debugging on the configuration of the web server, nginx located in /etc/nginx/ and uwsgi located in /etc/uwsgi/ directory. Reload the service by using **service** command (also with **sudo**), then recheck the log again.

```
sudo service uwsgi reload
```

Code that Crashes

- What is the common reason a program keeps on crashing? The reason is it tries to access invalid memory, meaning that the process attempts to access a portion of the system's memory that wasn't assigned to it. OS will raise an error like segmentation fault (**segfaults**) or general protection fault.
- To debug an application that's segfaulting, collect the **core file** which stored crash information, download the debugging symbols for that application, attach a debugger to it, and see where the fault occurs.
- **Valgrind** is a powerful tool that can tell if the code is doing any invalid operations, no matter if it crashes or not, available for Linux & Mac OS. There's similar tool called Dr. Memory for Windows & Linux.

Code that Crashes

- Some programming languages like Python, Java, or Ruby handle memory management to avoid accessing invalid memory. But in these languages there are still errors or exceptions triggered when a program comes across an unexpected condition that isn't correctly handled in the code.
- When those failures happen, the interpreter that's running the program will print the type of error, the line that caused the failure, and the traceback. The traceback shows the lines of the different functions that were being executed when the problem happened.
- If the error message isn't enough, use the debugging tools for the application's language. For Python, use **pdb** interactive debugger. Otherwise, use **printf debugging technique** to manually add statement that prints data related to the codes execution.

Code that Crashes

- Related to change code to print messages to the screen, the best approach to add the messages (in a way that can be easily enables or disabled) depends on whether process the debug info or not. In Python use the **logging** module.
- Fix the crashes problem by ensuring the program to be more resilient to failures. Instead of crashing unexpectedly, the program should inform the problem to the user and tell them what they need to do. Eg: modify the code to catch the error and tell the user the problem and how to fix it.

Code that Crashes

- Before fixing someone else's code, spend some time to understand what's going on with the code.
- Try reading the code's comments and documentation. This is a great place to start when trying to figure out what's going on. Otherwise, contribute to complete the comments to solidify the understanding of the workflow. Another thing that can help is reading the tests associated with the code.
- One approach to read some else's code would be to start with the function where the error happens, then the function or functions that call it, and so on until the context leading to the problem.

Code that Crashes

- Start the debugger by running pdb3 and then passing the script and parameters. Showing (Pdb) as the debugger prompt.
- Use **run** command to evaluate line by line or **continue** to run the script until it finishes or crashes.
- Inside the debugger, use **print** command to show the value of a variable.

Using Python Debugger pdb3

```
$ pdb3 update_products.py new_products.csv
…  # stripped output

(Pdb) continue
…  # stripped output
KeyError: 'product_code'
Uncaught exception. Entering post mortem
debugging
Running 'cont' or 'step' will restart the
program
-> row['product_code'], row['description']))

(Pdb) print(row)
OrderedDict([('\ufeffproduct_code', 'AV-101'),
… ])
(Pdb)
```

run until finish or crash

the error

byte order mark (BOM), so the key not found

Handing Bigger Incidents

- In the complex systems that involve many different services, you need to take a bird-eye view or a bigger picture of the problem and have different computers interact with each other.
- Some of the techniques you can use when facing a problem in a complex system are: looking at the available logs, figuring out what changed since the system was last working, rolling back to a previous state, removing faulty servers from the pool, or deploying new servers on demand.

Handing Bigger Incidents

- For a large incident and a big team working on figuring out the solution, there's an important roles:
  - **Communications Lead**, person in charge to communicating with the people affected. This person need to know what's going on and provide timely updates on the current state and how long until the problem is resolved.
  - **Incident Commander/Incident Controller**, one person in charge of delegating the different tasks to the team member. This person need to look at the big picture and decide what's the best use of the available resources.

Handing Bigger Incidents

- Postmortems are documents that describe details of incidents to help the team learn from the mistakes.
- Usually postmortems document breaks down what happened, why it happened, how it was diagnosed, how it was fixed, and finally figure out what the team can do to avoid the same event happening in the future.
- In general, postmortem includes the details of what caused the issue, what the impact of the issue was, how it got diagnosed, the short-term remediation applied, and the long-term remediation recommended. Last but not least, it is always useful to include what went well in the process.

Managing Computer Resources

- Memory leak happens when a chunk of memory that's no longer needed is not released. When the memory leaked becoming bigger and more complex over time, it can cause the whole system misbehaving.
- In the programming languages with memory management, there lies Garbage collector in charge or freeing the memory no longer in use.
- If there's a suspect program with a memory leak, use a memory profiler to figure out how the memory is being used. For example, Valgrind to profile C or C++ apps, or many profiler available for Python program.

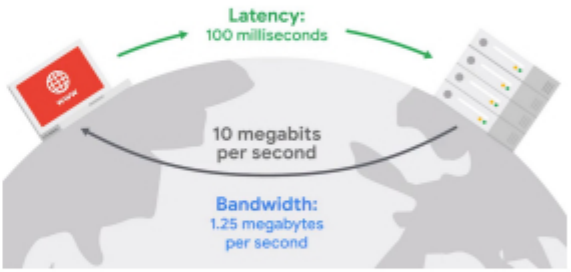
Managing Computer Resources

- Related to disk usage, if the resource is full, there will be some error message like no space left on device when running the applications or in the logs.
- Look at how space is being used and what directories taking up the most space, then drill down until to find out whether large chunks of space are taken by valid information or by files that should be perched.
- To check the currently opened files and filter out the ones that are deleted, use **lsof** command preceded with **sudo** to get superuser access.

```
sudo lsof | grep deleted
```

Managing Computer Resources

- The two most important factors that determine the time it takes to get the data over the network are the latency and the bandwidth. **Latency** is the delay between sending a byte of data from one point and receiving it on the other. **Bandwidth** is how much data can be sent or received in a second.
- For transmitting a lot of small pieces of data, it affects more on latency than bandwidth. On the flip side, for transmitting large chunks of data, if affects more on the bandwidth than the latency.



Managing Computer Resources

- In Linux, to check which processes are using the network connection is by running a program like **iftop**. It shows how much data each active connection is sending over the network.
- The more users sharing the same network, the slower the data comes in.
- Restrict how much each connection takes by using traffic shaping. This is a way of marking the data packets sent over the network with different priorities. The purpose is to avoid having huge chunks of data use all of the bandwidth.

Managing Computer Resources

- This Python code uses a module called **memory_profiler**, one of the many different profilers for Python.
- Add **@profile** label (decorator) before the function definition in order to tell the profiler to analyze the its memory consumption.
- **Decorator** used in Python to add extra behavior to functions without having to modify the code.

Dealing with Memory Leaks

```
from memory_profiler import profile

@profile
def main():
    words = {}
     # stripped out the function details
```

Managing Our Time

- Using the **Eisenhower Decision Matrix** to split tasks into two different categories: **urgent** and **important**.
- Example of important task that might not necessarily be urgent is solving **technical debt**. This is the pending work that accumulates when choosing a quick-and-easy solution over a sustainable long-term one.

	Urgent	Not urgent
Important	① ASAP	② Long term
Not important	③ Interruptions	④ Distractions

Managing Our Time

- Related to **interruptions** which are urgent but not important tasks, it's part of the IT support's role (to be interrupted). So, manage the interruptions effectively.
    - If working in a team, try to rotate the person to deal with those interruptions.
    - If working independently, try to establish a set of hours when user can expect to reach for normal requests and the rest of the time only be available for emergencies.

Managing Our Time

- Basic structure that can help to organize and prioritize tasks.
  1. Make a list of all the tasks that need to get done
  2. Check the real urgency of the tasks
     - Once it’s done with the most critically urgent task, take a look at the rest of the list and process to the next step.
  3. Assess the importance of each issue
  4. How much effort they'll take
     - One common technique is to divide them into small, medium, and large.

Managing Our Time

- Be realistic when trying to **estimate** how long it will take to complete a project, big or small. Avoid being overly optimistic with an estimated time.
- The best way to estimate is to compare the task with similar tasks done beforehand. It provides an estimate of how long similar projects actually took place in the past.
- Split the task that needs to be estimated into smaller steps then compare each step to a similar task done in the past. Based on that, assigns an estimate amount of time to each step.
- Once you have a rough estimate of the total time of all the steps, add a **factor** in some extra time for integration, which is also comes from prior experience.

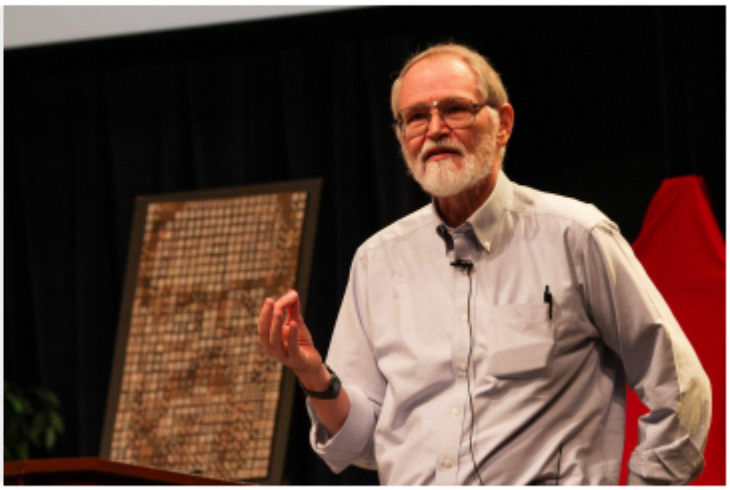
Managing Our Time

- Users will be happy if the issue (they raised) is resolved timely as expected. On the other hand, it will be frustrating if it takes much longer than they thought. But as long as there's an active and timely communication about such circumstances, they'll be able to understand this & manage their time.
- So as a general rule, communication is key. Try to be clear and upfront about when to expect the issue will be resolved. Should there be any difficult/pending issue to solve, explain why and what the next expectation should be.

Making Our Future Lives Easier

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will ever debug it?"

~ Brian W. Kernighan



* Image from https://commons.wikimedia.org/wiki/File:Brian_Kernighan_in_2012_at_Bell_Labs_1.jpg

Making Our Future Lives Easier

- If the code is clear and simple, it will be much easier to debug than if it's clever but obscure. The same applies to IT systems. If the system is engineered very cleverly, it will be extremely hard to understand what's going on with it when something fails.
- One valuable advice is to **develop code in small, digestible chunks**.
- Sometimes a change of scenery is all we need for a new idea to pop up and help us figure out what we're missing. And don't be afraid to ask for help, because sometimes just the act of explaining the problem to someone else can help us realize what's missing. Such simply technique is called **rubber duck debugging**, which is simply explaining the problem to a rubber duck. It sounds absurd, right? But yes it can really well.

Making Our Future Lives Easier

- When writing a code, it's important to make sure that the code has good unit tests and integrations tests. Other than manual and automatic tests, it needs to be able to easily roll back to the previous version.
- Make troubleshooting process easier by including good debug logging in the code. Also, with centralized logs collection, it minimizes the process to individually connect to each machine.
- Having a good monitoring and ticketing system can be helpful. The purpose is, to catch issues early before it affects too many users.
- Remember to spend time writing documentation. Just as importantly, store the documentation in a well-known location.

Making Our Future Lives Easier

- Spend some time to think and plan for additional resources when dealing with a service that's expected to grow and will acquire more resources in the future. Once the current usage and the expected growth is clearly figured out, don't forget to write it down as a reference in the future and check if anything has changed.
- Setting up service to run on the cloud will require some initial setup time, as well as an ongoing cost for the Cloud resources used. This is more expensive than running the service on-premise. Therefore, just basically delegate all capacity planning need to the Cloud provider.

Making Our Future Lives Easier

- One key strategy to prevent future problems is to make good use of **monitoring**. The computers send their data to a centralized location that aggregates all of the information, so it will be easy to check the condition or trigger alerts when the values are not within acceptable range.
- An important capability of monitoring is to include the measurements takes a long a period of time. That way, it can keep track of how the resource being used and catch changes in tendencies early on to help the planning process.


bangkit