- Venv:
- **Create a virtual environment for each project:** Whenever you start a new project, create a new virtual environment. This ensures a clean and isolated workspace.
- **Use requirements files:** To document and manage your project's dependencies, create a requirements.txt file. This file lists all the libraries and their versions. You can generate it using `pip freeze > requirements.txt` and later install them in a new environment using `pip install -r requirements.txt`.
- **Activate and deactivate:** Always activate the appropriate virtual environment before working on a project and deactivate it when you're done. This prevents confusion and potential conflicts.
- **Version control:** If you're collaborating with others, include the virtual environment setup instructions in your version control system. This ensures everyone is using the same environment.
- **Upgrade pip and setuptools:** When you create a new virtual environment, it's a good practice to upgrade `pip` and `setuptools` to the latest version. This ensures you're using the most up-to-date tools.

Benefits of automation:

- Scalability
- Centralizing mistakes
- Make debugging easier by logging the actions it takes
- Soft ROI can improve but difficult to measure
    - Team morale and stuff like that

Cons of automation:

- Is the time to create the script more than the benefits?
    - [time_to_automate < (time_to_perform * amount_of_times_done)]
    - If manually generating a daily report takes 5 mins, and the time to write the script takes 1 hour. Then after 12 days you’re saving time on this task.
- Bit-rot: process of software falling out of step with environment
    - New server added and all disk identifiers are changed by 1
        - Build a method of notifications to catch failures
        - Schedule restore data day. Can also be automated to check and compare

Pareto Principle

- 20% of sys admin tasks you perform are responsible for 80% of your work
- Try and identify and automate those 20% of your tasks

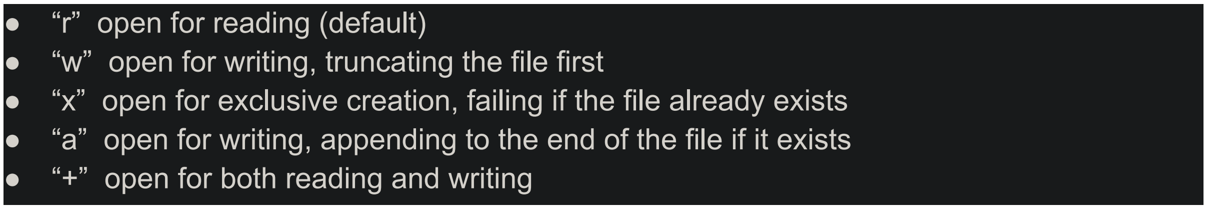
Important Modules

- Shutil
    - Disk_usage function
        - Get current available disk space
- Psutil
    - Cpu_percent function
        - Returns number of how much cpu is being used

Example Code

```
with open("spider.txt") as file:
    for line in file:
        print(line.strip().upper())

with open("novel.txt", "w") as file:
    file.write("It was a dark and stormy night")
```



- “r” open for reading (default)
- “w” open for writing, truncating the file first
- “x” open for exclusive creation, failing if the file already exists
- “a” open for writing, appending to the end of the file if it exists
- “+” open for both reading and writing

```
#Windows file directory written in Python
C:/my-directory/target-file.txt.
#CWD command for external files:
outputs['current_directory_before'] = os.getcwd()
outputs[‘files_and_directories’] = os.listdir()
outputs[‘path_value’] = os.environ.get(‘PATH’)

import os
os.remove("novel.txt")
os.rename("first_draft.txt", "finished_masterpiece.txt")
os.path.exists("finished_masterpiece.txt")
os.path.getsize("spider.txt")
os.path.getmtime("spider.txt")
import datetime
timestamp = os.path.getmtime("spider.txt")
datetime.datetime.fromtimestamp(timestamp)
os.path.abspath("spider.txt")

print(os.getcwd())
os.mkdir("new_dir")
os.rmdir("newer_dir")
os.chdir("new_dir")
os.getcwd()
import os
os.listdir("website")
dir = "website"
 for name in os.listdir(dir):
     fullname = os.path.join(dir, name)
     if os.path.isdir(fullname):
          print("{} is a directory".format(fullname))
     else:
          print("{} is a file".format(fullname))
```

https://www.coursera.org/learn/python-operating-system/supplement/aev5I/study-guide-files-and-directories

```
# Create a directory and move a file from one directory to another
# using low-level OS functions.

import os

# Check to see if a directory named "test1" exists under the current
# directory. If not, create it:
dest_dir = os.path.join(os.getcwd(), "test1")
if not os.path.exists(dest_dir):
 os.mkdir(dest_dir)


# Construct source and destination paths:
src_file = os.path.join(os.getcwd(), "sample_data", "README.md")
dest_file = os.path.join(os.getcwd(), "test1", "README.md")


# Move the file from its original location to the destination:
os.rename(src_file, dest_file)

# Create a directory and move a file from one directory to another
# using Pathlib.

from pathlib import Path

# Check to see if the "test1" subdirectory exists. If not, create it:
dest_dir = Path("./test1/")
if not dest_dir.exists():
  dest_dir.mkdir()

# Construct source and destination paths:
src_file = Path("./sample_data/README.md")
```

```
dest_file = dest_dir / "README.md"

# Move the file from its original location to the destination:
src_file.rename(dest_file)
```

CSV

```
import csv
 f = open("csv_file.txt")
 csv_f = csv.reader(f)
 for row in csv_f:
     name, phone, role = row
     print("Name: {}, Phone: {}, Role: {}".format(name, phone, role))
f.close()

import csv

hosts = [["workstation.local", "192.168.25.46"],["webserver.cloud",
"10.2.5.6"]]
with open('hosts.csv', 'w') as hosts_csv:
    writer = csv.writer(hosts_csv)
    writer.writerows(hosts)

cat software.csv
with open('software.csv') as software:
    reader = csv.DictReader(software)
    for row in reader:
      print(("{} has {} users").format(row["name"], row["users"]))

users = [ {"name": "Sol Mansi", "username": "solm", "department": "IT
infrastructure"},
 {"name": "Lio Nelson", "username": "lion", "department": "User Experience
Research"},
  {"name": "Charlie Grey", "username": "greyc", "department":
"Development"}]
keys = ["name", "username", "department"]
with open('by_department.csv', 'w') as by_department:
    writer = csv.DictWriter(by_department, fieldnames=keys)
    writer.writeheader()
    writer.writerows(users)
```

Regular Expressions

https://www.coursera.org/learn/python-operating-system/supplement/NVXqf/study-guide-regular-expressions

```
import re
result = re.search(r"aza", "maze")
print(result)

print(re.search(r"^x", "xenon"))

import re
print(re.search(r"[a-z]way", "The end of the highway"))
print(re.search(r"[a-z]way", "What a way to go"))
print(re.search("cloud[a-zA-Z0-9]", "cloudy"))
print(re.search("cloud[a-zA-Z0-9]", "cloud9"))

import re
print(re.search(r"[^a-zA-Z]", "This is a sentence with spaces."))
print(re.search(r"[^a-zA-Z ]", "This is a sentence with spaces."))
print(re.search(r"cat|dog", "I like cats."))
print(re.search(r"cat|dog", "I love dogs!"))
print(re.search(r"cat|dog", "I like both dogs and cats."))
print(re.search(r"cat|dog", "I like cats."))
print(re.search(r"cat|dog", "I love dogs!"))
print(re.search(r"cat|dog", "I like both dogs and cats."))
print(re.findall(r"cat|dog", "I like both dogs and cats."))
import re
print(re.search(r"Py.*n", "Pygmalion"))
print(re.search(r"Py.*n", "Python Programming"))
print(re.search(r"Py[a-z]*n", "Python Programming"))
print(re.search(r"Py[a-z]*n", "Pyn"))
import re
print(re.search(r"o+l+", "goldfish"))
print(re.search(r"o+l+", "woolly"))
print(re.search(r"o+l+", "boil"))
import re
print(re.search(r"p?each", "To each their own"))
print(re.search(r"p?each", "I like peaches"))
import re
print(re.search(r".com", "welcome"))
print(re.search(r"\.com", "welcome"))
print(re.search(r"\.com", "mydomain.com"))
import re
print(re.search(r"\w*", "This is an example"))
print(re.search(r"\w*", "And_this_is_another"))
```

\w is letters,numbers, and underscores

\d is digits

\s is space, tab, newline

\b is word boundaries and a few others

```
import re
print(re.search(r"A.*a", "Argentina"))
print(re.search(r"A.*a", "Azerbaijan"))
print(re.search(r"^A.*a$", "Australia"))
import re
pattern = r"^[a-zA-Z_][a-zA-Z0-9_]*$"
print(re.search(pattern, "_this_is_a_valid_variable_name"))
print(re.search(pattern, "this isn't a valid variable"))
print(re.search(pattern, "my_variable1"))
print(re.search(pattern, "2my_variable1"))
```

r”\d{3}-\d{3}-\d{4}” This line of code matches U.S. phone numbers in the format 111-222-3333.

r”^-?\d*(\.\d+)?$” This line of code matches any positive or negative number, with or without decimal places.

r”^(.+)\/([^\/]+)\/” This line of code matches any path and filename.

https://regex101.com/

```
import re
result = re.search(r"^(\w*), (\w*)$", "Lovelace, Ada")
print(result)
print(result.groups())
print(result[0])
print(result[1])
print(result[2])
"{} {}".format(result[2], result[1])
import re
def rearrange_name(name):
    result = re.search(r"^(\w*), (\w*)$", name)
    if result is None:
        return name
    return "{} {}".format(result[2], result[1])
rearrange_name("Lovelace, Ada")
import re
```

```
def rearrange_name(name):
    result = re.search(r"^([\w \.-]*), ([\w \.-]*)$", name)
    if result == None:
        return name
    return "{} {}".format(result[2], result[1])
rearrange_name("Hopper, Grace M.")
import re
print(re.findall(r"[a-zA-Z]{5}", "a scary ghost appeared"))
import re
print(re.findall(r"\w{5,10}", "I really like strawberries"))
import re
re.split(r"[.?!]", "One sentence. Another one? And the last one!")
import re
re.sub(r"[\w.%+-]+@[\w.-]+", "[REDACTED]", "Received an email for
go_nuts95@my.example.com")
import re
re.sub(r"^([\w .-]*), ([\w .-]*)$", r"\2 \1", "Lovelace, Ada")
```

https://www.coursera.org/learn/python-operating-system/supplement/fv5zk/study-guide-advanced-regular-expressions

Alteration: RegEx that matches any one of the alternatives separated by the pipe symbol

Backreference: This is applied when using re.sub( ) to substitute the value of a capture group into the output

Character classes: These are written inside square brackets and let us list the characters we want to match inside of those brackets

Character ranges: Ranges used to match a single character against a set of possibilities

grep: An especially easy to use yet extremely powerful tool for applying RegExes

Lookahead: RegEx that matches a pattern only if it’s followed by another pattern

Regular expression: A search query for text that's expressed by string pattern, also known as RegEx or RegExp

Wildcard: A character that can match more than one character

Read again about Extracting a PID using regexes in Python.

Data Streams

```
cat hello.py
#!/usr/bin/env python3
```

```
name = input("Please enter your name: ")
print("Hello, " + name)
```

```
def to_seconds(hours, minutes, seconds):
    return hours*3600+minutes*60+seconds

print("Welcome to this time converter")

cont = "y"
while(cont.lower() == "y"):
    hours = int(input("Enter the number of hours: "))
    minutes = int(input("Enter the number of minutes: "))
    seconds = int(input("Enter the number of seconds: "))

    print("That's {} seconds".format(to_seconds(hours, minutes, seconds)))
    print()
    cont = input("Do you want to do another conversion? [y to continue] ")

print("Goodbye!")
```

```
cat streams.py
#!/usr/bin/env python3

data = input("This will come from STDIN: ")
print("Now we write it to STDOUT: " + data)
print("Now we generate an error to STDERR: " + data + 1)
```

```
./streams.py
This will come from STDIN: Python Rocks!
Now we write it to STDOUT: Python Rocks!
```

```
cat greeting.txt
Well hello there, STDOUT
```

```
cat greeting.txt
Well hello there, STDOUT
```

```
ls -z
```

```
echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
cat variables.py
#!/usr/bin/env python3
import os
print("HOME: " + os.environ.get("HOME", ""))
print("SHELL: " + os.environ.get("SHELL", ""))
print("FRUIT: " + os.environ.get("FRUIT", ""))
./variables.py
export FRUIT=Pineapple
./variables.py

cat parameters.py
#!/usr/bin/env python3
import sys
print(sys.argv)

./parameters.py
['./parameters.py']

./parameters.py one two three
['./parameters.py', 'one', 'two', 'three']


wc variables.py
7 19 174 variables.py
echo $?
0

wc notpresent.sh
wc: notpresent.sh: No such file or directory
echo $?
1

#!/usr/bin/env python3

import os
import sys

filename = sys.argv[1]
```

```
if not os.path.exists(filename):
    with open(filename, "w") as f:
        f.write("New file created\n")
else:
    print("Error, the file {} already exists!".format(filename))
    sys.exit(1)
```

```
./create_file.py example
echo $?
0
```

```
cat example
New file created
./create_file.py example
Error, the file example already exists!
echo $?
1
```

```
>>> my_number = input('Please Enter a Number: \n')
Please Enter a Number:
123 + 1
>>> print(my_number)
123 + 1
>>> eval(my_number)
124
```

```
import subprocess
subprocess.run(["date"])
subprocess.run(["sleep", "2"])
result = subprocess.run(["ls", "this_file_does_not_exist"])
print(result.returncode)
```

```
result = subprocess.run(["host", "8.8.8.8"], capture_output=True)
```

```
result = subprocess.run(["host", "8.8.8.8"], capture_output=True)
print(result.returncode)
```

```
result = subprocess.run(["host", "8.8.8.8"], capture_output=True)
print(result.stdout)
```

```
result = subprocess.run(["host", "8.8.8.8"], capture_output=True)
print(result.stdout.decode().split())
```

```
import subprocess
result = subprocess.run(["rm", "does_not_exist"], capture_output=True)
```

```
import subprocess
result = subprocess.run(["rm", "does_not_exist"], capture_output=True)
print(result.returncode)
```

```
import subprocess
result = subprocess.run(["rm", "does_not_exist"], capture_output=True)
print(result.returncode)
print(result.stdout)
print(result.stderr)
```

```
import os
import subprocess

my_env = os.environ.copy()
my_env["PATH"] = os.pathsep.join(["/opt/myapp/", my_env["PATH"]])

result = subprocess.run(["myapp"], env=my_env)
```

https://www.coursera.org/learn/python-operating-system/supplement/hjJ5u/study-guide-python-subprocesses

Study guide: Python subprocesses

In Python, there are usually a lot of different ways to accomplish the same task. Some are easier to write, some are better suited to a given task, and some have a lower overhead in terms of the amount of computing power used. Subprocesses are a way to call and run other applications from within Python, including other Python scripts. In Python, the subprocess module can run new codes and applications by launching the new processes from the Python program. Because subprocess allows you to spawn new processes, it is a very useful way to run multiple processes in parallel instead of sequentially.

Python subprocess can launch processes to:

- Open multiple data files in a folder simultaneously.
- Run external programs.
- Connect to input, output, and error pipes and get return codes.

Comparing subprocess to OS and Pathlib

Again, Python has multiple ways to achieve most tasks; subprocess is extremely powerful, as it allows you to do anything you would from Python in the shell and get information back into Python. But just because you can use subprocess doesn’t always mean you'll want to.

Let’s compare subprocess to two of its alternatives: OS, which has been covered in other readings, and Pathlib. For tasks like getting the current working directory or creating a directory, OS and Pathlib are more direct (or “Pythonic,” meaning it uses the language as it was intended). Using subprocess for tasks like these is like using a crowbar to open a nut. It's more heavy-duty and can be overkill for simple operations.

As a comparison example, the following commands accomplish the exact same tasks of getting the current working directory.

Subprocess:

```
cwd_subprocess = subprocess.check_output(['pwd'], text=True).strip()
```

OS:

```
cwd_os = os.getcwd()
```

Pathlib:

```
cwd_pathlib = Path.cwd()
```

And these following commands accomplish the exact same tasks of creating a directory.

Subprocess:

```
subprocess.run(['mkdir', 'test_dir_subprocess2'])
```

OS:

```
os.mkdir('test_dir_os2')
```

Pathlib:

```
test_dir_pathlib2 = Path('test_dir_pathlib2')
test_dir_pathlib2.mkdir(exist_ok=True) #Ensures the directory is created only if it doesn't already exist
```

When to use subprocess

Subprocess is best used when you need to interface with external processes, run complex shell commands, or need precise control over input and output. Subprocess also spawns fewer processes per task than OS, so subprocess can use less compute power.

Other advantages include:

- Subprocess can run any shell command, providing greater flexibility.
- Subprocess can capture `stdout` and `stderr` easily.

On the other hand, OS is useful for basic file and directory operations, environment variable management, and when you don't need the object-oriented approach provided by Pathlib.

Other advantages include:

- OS provides a simple way to interface with the operating system for basic operations.

- OS is part of the standard library, so it's widely available.

Finally, Pathlib is most helpful for working extensively with file paths, when you want an object-oriented and intuitive way to handle file system tasks, or when you're working on code where readability and maintainability are crucial.
Other advantages include:

- Pathlib provides an object-oriented approach to handle file system paths.
- Compared to OS, Pathlib is more intuitive for file and directory operations.
- Pathlib is more readable for path manipulations.

Where subprocess shines

The basic ways of using subprocess are the **`.run()`** and **`.Popen()`** methods. There are additional methods, **`.call()`**, **`.check_output()`**, and **`.check_call()`**. Usually, you will just want to use **`.run()`** or one of the two check methods when appropriate. However, when spawning parallel processes or communicating between subprocesses, **`.Popen()`** has a lot more power!
You can think of **`.run()`** as the simplest way to run a command—it's all right there in the name—and **`.Popen()`** as the most fully featured way to call external commands.
All of the methods, **`.run()`**, **`.call()`**, **`.check_output()`**, and **`.check_call()`** are wrappers around the **`.Popen()`** class.

Run

The **`.run()`** command is the recommended approach to invoking subprocesses. It runs the command, waits for it to complete, then returns a CompletedProcess instance that contains information about the process.
Using **`.run()`** to execute the echo command:

```
result_run = subprocess.run(['echo', 'Hello, World!'], capture_output=True, text=True)
result_run.stdout.strip()  # Extracting the stdout and stripping any extra whitespace
```

output:

```
'Hello, World!'
```

Call

The **`call()`** command runs a command, waits for it to complete, then returns the return code. Call is older and **`.run()`** should be used now, but it's good to see how it works.
Using **`call()`** to execute the echo command:

```
return_code_call = subprocess.call(['echo', 'Hello from call!'])
return_code_call
```

output:

```
0
```

The returned value 0 indicates that the command was executed successfully.

Check_call and check_output

Use `check_call()` to receive just the status of a command. Use `check_output()` to also obtain output. These are good for situations such as file IO, where a file might not exis, or the operation may otherwise fail.

The command `check_call()`is similar to `call()` but raises a CalledProcessError exception if the command returns a non-zero exit code.

Using `check_call()` to execute the echo command:

```
return_code_check_call = subprocess.check_call(['echo', 'Hello from check_call!'])
return_code_check_call
```

output:

```
0
```

The returned value 0 indicates that the command was executed successfully.

Using `check_output()` to execute the echo command:

```
output_check_output = subprocess.check_output(['echo', 'Hello from check_output!'], text=True)
output_check_output.strip()  # Extracting the stdout and stripping any extra whitespace
```

output:

```
'Hello from check_output!'
```

Note: `Check_output` raises a CalledProcessError if the command returns a non-zero exit code. For more on CalledProcessError, see Exceptions.

Popen

`Popen()` offers more advanced features compared to the previously mentioned functions. It allows you to spawn a new process, connect to its input/output/error pipes, and obtain its return code.

Using Popen to execute the echo command:

```
process_popen = subprocess.Popen(['echo', 'Hello from popen!'], stdout=subprocess.PIPE, text=True)
output_popen, _ = process_popen.communicate()
output_popen.strip()  # Extracting the stdout and stripping any extra whitespace
```

output:

```
'Hello from popen!'
```

Pro tip

The Popen command is very useful when you need asynchronous behavior and the ability to pipe information between a subprocess and the Python program that ran that subprocess. Imagine you want to start a long-running command in the background and then continue with other tasks in your script. Later on, you want to be able to check if the process has finished. Here's how you would do that using Popen.

```
import subprocess
```

Using Popen for asynchronous behavior:

```
process = subprocess.Popen(['sleep', '5'])
message_1 = "The process is running in the background..."
# Give it a couple of seconds to demonstrate the asynchronous behavior
```

```
import time
time.sleep(2)
# Check if the process has finished
if process.poll() is None:
      message_2 = "The process is still running."
else:
      message_2 = "The process has finished."
print(message_1, message_2)
```

output:

```
('The process is running in the background...',
 'The process is still running.')
```

The process runs in the background as the script continues with other tasks (in this case, simply waiting for a couple of seconds). Then the script checks if the process is still running. In this case, the check was after 2 seconds' sleep, but Popen called sleep on 5 seconds. So the program confirms that the subprocess has not finished running.

Key takeaways

Subprocess is a powerful module that allows you to do anything you could in Python from within the shell, then get information back into Python. You'll probably want to stick with OS for basic file and directory operations or Pathlib for working extensively with file paths. But when you interface with external processes, run complex shell commands, or need precise control over input and output, the subprocess module is the way to go.

```
#!/bin/env/python3

import sys

logfile = sys.argv[1]
with open(logfile) as f:
  for line in f:
    print(line.strip())
```

```
#!/bin/env/python3

import sys

logfile = sys.argv[1]
with open(logfile) as f:
  for line in f:
    if "CRON" not in line:
      continue
    print(line.strip())
```

```
import re
pattern = r"USER \((\w+)\)$"
line = "Jul 6 14:03:01 computer.name CRON[29440]: USER (naughty_user)"
result = re.search(pattern, line)
print(result[1])
```

```
#!/bin/env/python3

import re
import sys

logfile = sys.argv[1]
with open(logfile) as f:
  for line in f:
    if "CRON" not in line:
      continue
    pattern = r"USER \((.+)\)$"
    result = re.search(pattern, line)
    print(result[1])
```

```
chmod +x check_cron.py
./check_cron.py syslog
```

```
usernames = {}
name = "good_user"
usernames[name] = usernames.get(name, 0) + 1
print(usernames)
usernames[name] = usernames.get(name, 0) + 1
print(usernames)
```

```
#!/bin/env/python3

import re
import sys

logfile = sys.argv[1]
usernames = {}
with open(logfile) as f:
  for line in f:
    if "CRON" not in line:
      continue
    pattern = r"USER \((\w+)\)$"
    result = re.search(pattern, line)

    if result is None:
      continue
    name = result[1]
    usernames[name] = usernames.get(name, 0) + 1

print(usernames)
./check_cron.py syslog
```

Terms and definitions from Course 2, Module 4

Bash: The most commonly used shell on Linux
Command line arguments: Inputs provided to a program when running it from the command line
Environment variables: Settings and data stored outside a program that can be accessed by it to alter how the program behaves in a particular environment
Input / Output (I/O): These streams are the basic mechanism for performing input and output operations in your programs
Log files: Log files are records or text files that store a history of events, actions, or errors generated by a computer system, software, or application for diagnostic, troubleshooting, or auditing purposes
Standard input stream commonly (STDIN): A channel between a program and a source of input
Standard output stream (STDOUT): A pathway between a program and a target of output, like a display
Standard error (STDERR): This displays output like standard out, but is used specifically as a channel to show error messages and diagnostics from the program
Shell: The application that reads and executes all commands
Subprocesses: A process to call and run other applications from within Python, including other Python scripts

unittest

A unittest provides developers with a set of tools to construct and run tests. These tests can be run on individual components or by isolating units of code to ensure their correctness. By running unittests, developers can identify and fix any bugs that appear, creating a more reliable code. In this reading, you will learn about unittest concepts, how to use and when to use them, and view an example along the way.

Concepts

Unittest relies on the following concepts:

- **Test fixture:** This refers to preparing to perform one or more tests. In addition, test fixtures also include any actions involved in testing cleanup. This could involve creating temporary or proxy databases, directories, or starting a server process.
- **Test case:** This is the individual unit of testing that looks for a specific response to a set of inputs. If needed, TestCase is a base class provided by unittest and can be used to create new test cases.
- **Test suite:** This is a collection of test cases, test suites, or a combination of both. It is used to compile tests that should be executed together.
- **Test runner:** This runs the test and provides developers with the outcome's data. The test runner can use different interfaces, like graphical or textual, to provide the developer with the test results. It can also provide a special value to developers to communicate the test results.

Use case

Let's look at a test case example where the Python code simulates a cake factory and performs different functions. These include choosing different sizes and flavors of a cake, including small, medium, and large, and chocolate or vanilla. In addition, the simple class allows developers to add sprinkles or cherries to the cake, return a list of ingredients, and return the price of the cake based on size and toppings. Run the following code:

```python
from typing import List


class CakeFactory:
 def __init__(self, cake_type: str, size: str):
   self.cake_type = cake_type
   self.size = size
   self.toppings = []

   # Price based on cake type and size
   self.price = 10 if self.cake_type == "chocolate" else 8
   self.price += 2 if self.size == "medium" else 4 if self.size == "large"
else 0
```

```
 def add_topping(self, topping: str):
     self.toppings.append(topping)
     # Adding 1 to the price for each topping
     self.price += 1

 def check_ingredients(self) -> List[str]:
     ingredients = ['flour', 'sugar', 'eggs']
     ingredients.append('cocoa') if self.cake_type == "chocolate" else
ingredients.append('vanilla extract')
     ingredients += self.toppings
     return ingredients

 def check_price(self) -> float:
     return self.price

# Example of creating a cake and adding toppings
cake = CakeFactory("chocolate", "medium")
cake.add_topping("sprinkles")
cake.add_topping("cherries")
cake_ingredients = cake.check_ingredients()
cake_price = cake.check_price()


cake_ingredients, cake_price

import unittest

class TestCakeFactory(unittest.TestCase):
 def test_create_cake(self):
   cake = CakeFactory("vanilla", "small")
   self.assertEqual(cake.cake_type, "vanilla")
   self.assertEqual(cake.size, "small")
   self.assertEqual(cake.price, 8) # Vanilla cake, small size

 def test_add_topping(self):
     cake = CakeFactory("chocolate", "large")
     cake.add_topping("sprinkles")
     self.assertIn("sprinkles", cake.toppings)

 def test_check_ingredients(self):
     cake = CakeFactory("chocolate", "medium")
     cake.add_topping("cherries")
     ingredients = cake.check_ingredients()
     self.assertIn("cocoa", ingredients)
     self.assertIn("cherries", ingredients)
     self.assertNotIn("vanilla extract", ingredients)

 def test_check_price(self):
     cake = CakeFactory("vanilla", "large")
     cake.add_topping("sprinkles")
     cake.add_topping("cherries")
     price = cake.check_price()
     self.assertEqual(price, 13) # Vanilla cake, large size + 2 toppings


# Running the unittests
unittest.TextTestRunner().run(unittest.TestLoader().loadTestsFromTestCase(TestCakeFactory))
..F.
======================================================================
FAIL: test_check_price (__main__.TestCakeFactory)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-9-32dbf74b3655>", line 33, in test_check_price
    self.assertEqual(price, 13) # Vanilla cake, large size + 2 toppings
AssertionError: 14 != 13

----------------------------------------------------------------------
Ran 4 tests in 0.007s

FAILED (failures=1)
<unittest.runner.TextTestResult run=4 errors=0 failures=1>
```

pytest

Pytest is a powerful Python testing tool that assists programmers in writing more effective and stable programs. It helps to simplify the process of writing, organizing and executing tests. It can be used to write a variety of tests including: integration, end-to-end, and functional tests. It supports automatic test discovery and generates informative test reports.

In this reading, you will learn more about pytests, how to write tests with pytest, and its fixtures.

How to write tests

Pytests are written with functions that use the operation, **`assert()`**. An assert is a commonly used debugging tool in Python that allows programmers to include sanity checks in their code. They ensure certain conditions or assumptions hold true during runtime. If the condition provided to **`assert()`** turns out to be false, it indicates a bug in the code, an exception is raised, and halts the program's execution. Typically, code provides an assert condition followed by an optional message. An example is:

```python
def divide(a, b):
    assert b != 0, "Cannot divide by zero"
    return a / b
```

Pytest fixtures

Fixtures are used to separate parts of code that only run for tests. They are reusable pieces of test setups and teardown code that are shared across multiple tests. Fixtures benefit developers by assisting in keeping their tests clean and avoiding code duplication. Let's look at an example of using a pytest in Python:

```python
import pytest
class Fruit:
    def __init__(self, name):
        self.name = name
        self.cubed = False

    def cube(self):
        self.cubed = True

class FruitSalad:
    def __init__(self, *fruit_bowl):
        self.fruit = fruit_bowl
        self._cube_fruit()

    def _cube_fruit(self):
        for fruit in self.fruit:
            fruit.cube()

# Arrange
```

```
@pytest.fixture
def fruit_bowl():
    return [Fruit("apple"), Fruit("banana")]


def test_fruit_salad(fruit_bowl):
    # Act
    fruit_salad = FruitSalad(*fruit_bowl)


    # Assert
    assert all(fruit.cubed for fruit in fruit_salad.fruit)
```

In this example, `test_fruit_salad` requests `fruit_bowl`. When pytest recognizes this, it executes the `fruit_bowl` fixture function and takes the object it returns into `test_fruit_salad` as the `fruit_bowl` argument.

Comparing unittest and pytest

Both `unittest` and `pytest` provide developers with tools to create robust and reliable code through different forms of tests. Both can be used while creating programs within Python, and it is the developer's preference on which type they want to use.

In this reading, you will learn about the differences between `unittest` and `pytest`, and when to use them.

Key differences

`Unittest` is a tool that is built directly into Python, while `pytest` must be imported from outside your script. Test discovery acts differently for each test type. `Unittest` has the functionality to automatically detect test cases within an application, but it must be called from the command line. `Pytests` are performed automatically using the prefix `test_`. `Unittests` use an object-oriented approach to write tests, while `pytests` use a functional approach. `Pytests` use built-in assert statements, making tests easier to read and write. On the other hand, `unittests` provide special assert methods like `assertEqual()` or `assertTrue()`.

Backward compatibility exists between `unittest` and `pytest`. Because `unittest` is built directly into Python, these test suites are more easily executed. But that doesn't mean that `pytest` cannot be executed. Because of backward compatibility, the `unittest` framework can be seamlessly executed using the `pytest` framework without major modifications. This allows developers to adopt `pytest` gradually and integrate them into their code.

```
#!/usr/bin/env python3
import re
def rearrange_name(name):
  result = re.search(r"^([\w .]*), ([\w .]*)$", name)
  return "{} {}".format(result[2], result[1])
from rearrange import rearrange_name
```

```
rearrange_name("Lovelace, Ada")
#!/usr/bin/env python3

import re

def rearrange_name(name):
  result = re.search(r"^([\w .]*), ([\w .]*)$", name)
  return "{} {}".format(result[2], result[1])
```

```
#!/usr/bin/env python3

import unittest

from rearrange import rearrange_name

class TestRearrange(unittest.TestCase):

  def test_basic(self):
    testcase = "Lovelace, Ada"
    expected = "Ada Lovelace"
    self.assertEqual(rearrange_name(testcase), expected)

# Run the tests
unittest.main()
```

```
chmod +x rearrange_test.py
./rearrange_test.py
```

Edge Cases

```
def test_empty(self):
  testcase = ""
  expected = ""
  self.assertEqual(rearrange_name(testcase), expected)

#!/usr/bin/env python3
```

```
import re

def rearrange_name(name):
  result = re.search(r"^([\w .-]*), ([\w .-]*)$", name)
  if result is None:
    return ""
  return "{} {}".format(result[2], result[1])

from rearrange import rearrange_name
import unittest

class TestRearrange(unittest.TestCase):

  def test_basic(self):
    testcase = "Lovelace, Ada"
    expected = "Ada Lovelace"
    self.assertEqual(rearrange_name(testcase), expected)

  def test_empty(self):
    testcase = ""
    expected = ""
    self.assertEqual(rearrange_name(testcase), expected)

  def test_double_name(self):
    testcase = "Hopper, Grace M."
    expected = "Grace M. Hopper"
    self.assertEqual(rearrange_name(testcase), expected)

  def test_one_name(self):
    testcase = "Voltaire"
    expected = "Voltaire"
    self.assertEqual(rearrange_name(testcase), expected)

# Run the tests
unittest.main()

import re

def rearrange_name(name):
  result = re.search(r"^([\w .]*), ([\w .]*)$", name)
  if result is None:
    return name
  return "{} {}".format(result[2], result[1])
```

You’ve learned that unit tests are designed to test small pieces of code, like a single function or method, to ensure that each part of the code is working as it should. Unit testing helps to isolate errors so bugs can be identified and fixed earlier on during the software development process before they can become larger, more expensive issues to fix.

You’ve also learned about the object-oriented concepts of **`unittest`**, a unit testing framework in Python that developers can use to help test their code. In this reading, you’ll learn more about test cases, running unit tests using the command-line interface, unit test design patterns, and some common, basic assertions that you can use when developing your own unit tests.

Test cases

The building blocks of unit tests within the **`unittest`** module are test cases, which enable developers to run multiple tests at once. To write test cases, developers need to write subclasses of **`TestCase`** or use **`FunctionTestCase`**.

To perform a specific test, the **`TestCase`** subclass needs to implement a test method that starts with the name **`test`**. This identifier is what informs the test runner about which methods represent tests.

Examine the following example for test cases:

```
import unittest


class TestStringMethods(unittest.TestCase):


    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')


    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())


    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)


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

- The assertEqual(a, b) method checks that a == b
- The assertNotEqual(a, b) method checks that a != b
- The assertTrue(x) method checks that bool(x) is True
- The assertFalse(x) method checks that bool(x) is False
- The assertIs(a, b) method checks that a is b
- The assertIsNot(a, b) method checks that a is not b
- The assertIsNone(x) method checks that x is None
- The assertIsNotNone(x) method checks that x is not None
- The assertIn(a, b) method checks that a in b
- The assertNotIn(a, b) method checks that a not in b
- The assertIsInstance(a, b) method checks that isinstance(a, b)
- The assertNotIsInstance(a, b) method checks that not isinstance(a,
-

To call an entire module:

```
python -m unittest test_module1 test_module2
```

To call a test class:

```
python -m unittest test_module.TestClass
```

To call a test method:

```
python -m unittest test_module.TestClass.test_method
```

Test modules can also be called using a file path, as written below:

```
python -m unittest tests/test_something.py
```

[source: https://docs.python.org/3/library/unittest.html]

Unit test design patterns

One pattern that you can use for unit tests is made up of three phases: arrange, act, and assert. Arrange represents the preparation of the environment for testing; act represents the action, or the objective of the test, performed; and assert represents whether the results checked are expected or not.

Imagine building a system for a library. The objective is to test whether a new book can be added to the library's collection and then to check if the book is in the collection.

Using the above structure of arrange, act, and assert, consider the following example code:

- What's given (arrange): A library with a collection of books
- When to test (act): A new book is added to the collection
- Then check (assert): The new book should be present in the library's collection

```
class Library:
    def __init__(self):
        self.collection = []

    def add_book(self, book_title):
        self.collection.append(book_title)

    def has_book(self, book_title):
        return book_title in self.collection

# Unit test for the Library system
class TestLibrary(unittest.TestCase):

    def test_adding_book_to_library(self):
        # Arrange
        library = Library()
        new_book = "Python Design Patterns"

        # Act
        library.add_book(new_book)

        # Assert
        self.assertTrue(library.has_book(new_book))

# Running the test
library_test_output =
unittest.TextTestRunner().run(unittest.TestLoader().loadTestsFromTestCase(
TestLibrary))
print(library_test_output)
```

Test suites

Testing can be time-intensive, but there are ways that you can optimize the testing process. The following methods and modules allow you to define instructions that execute before and after each test method:

- **`setUp()`** can be called automatically with every test that's run to set up code.
- **`tearDown()`** helps clean up after the test has been run.

If setUp()raises an exception during the test, the unittest framework considers this to be an error and the test method is not executed. If **`setUp()`** is successful, **`tearDown()`** runs even if the test method fails. You can add these methods to your unit tests, which you can then include in a test suite. Test suites are collections of tests that should be executed together—so all of the topics covered in this reading can be included within a test suite.

Consider the following code example to see how each of these unit testing components is used together and run within a test suite:

```python
import unittest
import os
import shutil

# Function to test
def simple_addition(a, b):
    return a + b

# Paths for file operations
ORIGINAL_FILE_PATH = "/tmp/original_test_file.txt"
COPIED_FILE_PATH = "/mnt/data/copied_test_file.txt"

# Global counter
COUNTER = 0

# This method will be run once before any tests or test classes
def setUpModule():
    global COUNTER
    COUNTER = 0

    # Create a file in /tmp
    with open(ORIGINAL_FILE_PATH, 'w') as file:
        file.write("Test Results:\n")

# This method will be run once after all tests and test classes
def tearDownModule():
    # Copy the file to another directory
    shutil.copy2(ORIGINAL_FILE_PATH, COPIED_FILE_PATH)
    # Remove the original file
    os.remove(ORIGINAL_FILE_PATH)

class TestSimpleAddition(unittest.TestCase):

    # This method will be run before each individual test
    def setUp(self):
        global COUNTER
        COUNTER += 1

    # This method will be run after each individual test
    def tearDown(self):
        # Append the test result to the file
        with open(ORIGINAL_FILE_PATH, 'a') as file:
            result = "PASSED" if self._outcome.success else "FAILED"
            file.write(f"Test {COUNTER}: {result}\n")

    def test_add_positive_numbers(self):
        self.assertEqual(simple_addition(3, 4), 7)

    def test_add_negative_numbers(self):
        self.assertEqual(simple_addition(-3, -4), -7)

# Running the tests
suite = unittest.TestLoader().loadTestsFromTestCase(TestSimpleAddition)
runner = unittest.TextTestRunner()
runner.run(suite)

# Read the copied file to show the results
with open(COPIED_FILE_PATH, 'r') as result_file:
    test_results = result_file.read()

print(test_results)
```

Type of Tests

- Unit tests - test individual functions
- Integration tests - test how each part of a system interacts with each other like with the api and database
- Regression tests - variant of unit test to verify that a bug has been fixed before trying to test it. Write a test that purposely fails to trigger the bug

- Smoke tests/build verification tests - Basic questions like - Does the program run - Check that service is running on certain port
- Load tests - Verify system works under big load. DDOS yourself
- Test Suite - A group of tests of one or many kinds
- Test Driven Dev - Create tests before writing code - Write test to make sure it fails > write the code to satisfy the test
- Continuous Integration - When you submit code to a repository it automatically goes through a test suite - version control

Try - Except & Raising Errors

```
#!/usr/bin/env python3

def character_frequency(filename):
  """Counts the frequency of each character in the given file."""
  # First try to open the file
  try:
    f = open(filename)
  except OSError:
    return None

  # Now process the file
  characters = {}
  for line in f:
    for char in line:
      characters[char] = characters.get(char, 0) + 1
  f.close()
  return characters
```

```
#!/usr/bin/env python3

def validate_user(username, minlen):
  if minlen < 1:
    raise ValueError("minlen must be at least 1")

  if len(username) < minlen:
    return False
  if not username.isalnum():
    return False
  return True
```

```
from validations import validate_user
validate_user("", -1)
from validations import validate_user
validate_user("", 1)
validate_user("myuser", 1)
from validations import validate_user
validate_user(88, 1)
from validations import validate_user
validate_user([], 1)
from validations import validate_user
validate_user(["name"], 1)
#!/usr/bin/env python3

def validate_user(username, minlen):
  assert type(username) == str, "username must be a string"
  if minlen < 1:
    raise ValueError("minlen must be at least 1")

  if len(username) < minlen:
    return False
  if not username.isalnum():
    return False
  return True
from validations import validate_user
validate_user([3], 1)

#!/usr/bin/env python3

import unittest

from validations import validate_user

class TestValidateUser(unittest.TestCase):
  def test_valid(self):
    self.assertEqual(validate_user("validuser", 3), True)

  def test_too_short(self):
    self.assertEqual(validate_user("inv", 5), False)

  def test_invalid_characters(self):
    self.assertEqual(validate_user("invalid_user", 1), False)
  def test_invalid_minlen(self):
    self.assertRaises(ValueError, validate_user, "user", -1)


# Run the tests
unittest.main()
```

Study guide: Handling errors

You've learned that in some cases, it's better to raise an error yourself, and how to test that the right error is raised when that's what you expect. You've also learned how to test your code to verify that it does what it should. In this reading, you'll learn about error handling syntax, including raising exceptions, using an assert statement, and the try and except clauses.

Exception handling

When performing exception handling, it is important to predict which exceptions can happen. Sometimes, to figure out which exceptions you need to account for, you have to let your program fail.

The simplest way to handle exceptions in Python is by using the try and except clauses.

In the **`try`** clause, Python executes all statements until it encounters an exception. You use the **`except`** clause to catch and handle the exception(s) that Python encounters in the **`try`** clause.

Here is the process for how it works:

1. Python runs the **`try`** clause, e.g., the statement(s) between the **`try`** and **`except`** keywords.
2. If no error occurs, Python skips the except clause and the execution of the **`try`** statement is finished.
3. If an error occurs during execution of the try clause, Python skips the rest of the try clause and transfers control to the corresponding except block. If the type of error matches what is listed after the except keyword, Python executes the except clause. The execution then continues on after the try/except block.
4. If an exception occurs but it does not match what is listed in the **`except`** clause, it is passed onto **`try`** statements outside of that **`try`**/**`except`** block. However, if a handler for that exception cannot be found, the exception becomes an unhandled exception, the execution stops, and Python displays a designated error message.

Sometimes, a **`try`** statement can have more than one **`except`** clause so that the code can specify handlers for different exceptions. This can help to reduce the number of unhandled exceptions.

You can use exceptions to catch almost everything. It is good practice as a developer or programmer to be as specific as possible with the types of exceptions that you intend to handle, especially if you're creating your own exceptions.

Raise exceptions

As a developer or programmer, you might want to raise an error yourself. Usually, this happens when some of the conditions necessary for a function to do its job properly aren't met and returning none or some other base value isn't good enough. You can raise an error or raise an exception (also known as “throwing an exception”), which forces a particular exception to occur, and notifies you that something in your code is going wrong or an error has occurred.

Here are some instances where raising an exception is a useful tool:

- A file doesn’t exist
- A network or database connection fails
- Your code receives invalid input

In the example below, the code raises two built-in Python exceptions: **`raise ValueError`** and raise **`ZeroDivisionError`**. You can find more information on these raises in the example below, along with explanations of potential errors that may occur during an exception.

Example exception handling

Now that you have an understanding of **`try`** and **`except`** clauses, **`assert`** statements, and raising exceptions, consider the following code examples which use all of these concepts together.

The basic structure of exception handling is as follows:

```python
# File reading function with exception handling
def read_file(filename):
    try:
        with open(filename, 'r') as f:
            return f.read()
    except FileNotFoundError:
        return "File not found!"
    finally:
        print("Finished reading file.")
def faulty_read_and_divide(filename):
    with open(filename, 'r') as file:
        data = file.readlines()
        num1 = int(data[0])
        num2 = int(data[1])
        return num1 / num2
```

There are several potential issues here:

- The file might not exist, causing a **FileNotFoundError**.
- The file might not have enough lines of data, leading to an **IndexError**.

- The data in the file might not be convertible to integers, raising a **ValueError**.
- The second number might be zero, which would raise a **ZeroDivisionError**.

To address these potential issues, you can add the appropriate exception handling illustrated below

```
def enhanced_read_and_divide(filename):
    try:
        with open(filename, 'r') as file:
            data = file.readlines()

        # Ensure there are at least two lines in the file
        if len(data) < 2:
            raise ValueError("Not enough data in the file.")

        num1 = int(data[0])
        num2 = int(data[1])

        # Check if second number is zero
        if num2 == 0:
            raise ZeroDivisionError("The denominator is zero.")

        return num1 / num2


    except FileNotFoundError:
             return "Error: The file was not found."
    except ValueError as ve:
             return f"Value error: {ve}"
    except ZeroDivisionError as zde:
             return f"Division error: {zde}"
```

Now, the function **enhanced_read_and_divide** is equipped to handle potential exceptions gracefully, providing informative error messages to the caller. This way, the code will explain when it fails since you have identified potential fault zones such as when dealing with unpredictable inputs or file content.

Notice how the exceptions are instantiated as objects (such as **ValueError ve**) that you can use to further diagnose the issue by printing them out. The errors should read:

```
File-level issues:
Value error: Not enough data in the file.
Error: The file was not found.
Data-level issues:
Value error: invalid literal for int() with base 10: 'apple'
Division error: The denominator is zero.
```

assert statements

assert statements help you to verify if a certain condition is met and throw an exception if it isn’t. As is stated in the name, their purpose is to "assert" that certain conditions are true at specific points in your program.

The **assert** statement exists in almost every programming language and has two main uses:

- To help detect problems earlier in development, rather than later when some other operation fails. Problems that aren’t addressed until later in the development process can turn out to be more time-intensive and costly to fix.
- To provide a form of documentation for other developers reading the code.

Terms and definitions from Course 2, Module 5

Automatic testing: A process where software checks itself for errors and confirms that it works correctly

Black-box tests: A test where there is an awareness of what the program is supposed to do but not how it does it

Edge cases: Inputs to code that produce unexpected results, found at the extreme ends of the ranges of input

Pytest: A powerful Python testing tool that assists programmers in writing more effective and stable programs

Software testing: A process of evaluating computer code to determine whether or not it does what is expected

Test case: This is the individual unit of testing that looks for a specific response to a set of inputs

Test fixture: This prepared to perform one or more tests

Test suite: This is used to compile tests that should be executed together

Test runner: This runs the test and provides developers with the outcome’s data

unittest: A set of Python tools to construct and run unit tests
Unit tests: A test to verify that small isolated parts of a program work correctly
White-box test: A test where test creator knows how the code works and can write test cases that use the understanding to make sure it performs as expected

Linux Shell & Bash - Speedrun

```
mkdir mynewdir
cd mynewdir/
/mynewdir$ pwd
/mynewdir$ cp ../spider.txt .
/mynewdir$ touch myfile.txt
/mynewdir$ ls -l
#Output:
#-rw-rw-r-- 1 user user   0 Mai 22 14:22 myfile.txt
#-rw-rw-r-- 1 user user 192 Mai 22 14:18 spider.txt
/mynewdir$ ls -la
#Output:
#total 12
#drwxr-xr-x  2 user user  4096 Mai 22 14:17 .
#drwxr-xr-x 56 user user 12288 Mai 22 14:17 ..
#-rw-rw-r--  1 user user     0 Mai 22 14:22 myfile.txt
#-rw-rw-r--  1 user user   192 Mai 22 14:18 spider.txt
/mynewdir$ mv myfile.txt emptyfile.txt
/mynewdir$ cp spider.txt yetanotherfile.txt
/mynewdir$ ls -l
#Output:
#total 8
#-rw-rw-r-- 1 user user   0 Mai 22 14:22 emptyfile.txt
#-rw-rw-r-- 1 user user 192 Mai 22 14:18 spider.txt
#-rw-rw-r-- 1 user user 192 Mai 22 14:23 yetanotherfile.txt
/mynewdir$ rm *
/mynewdir$ ls -l
#total 0
/mynewdir$ cd ..
rmdir mynewdir/
ls mynewdir
#ls: cannot access 'mynewdir': No such file or directory
```

```
cat stdout_example.py
#!/usr/bin/env python3
print("Don't mind me, just a bit of text here...")
./stdout_example.py
#Output: Don't mind me, just a bit of text here...
./stdout_example.py > new_file.txt
cat new_file.txt
#Output: Don't mind me, just a bit of text here...
./stdout_example.py >> new_file.txt
cat new_file.txt
#Output: Don't mind me, just a bit of text here...
 #Don't mind me, just a bit of text here...
cat streams_err.py
#!/usr/bin/env python3

data = input("This will come from STDIN: ")
print("Now we write it to STDOUT: " + data)
raise ValueError("Now we generate an error to STDERR")
./streams_err.py < new_file.txt
#This will come from STDIN: Now we write it to STDOUT: Don't mind #me,
just a bit of text here...
#Traceback (most recent call last):
  #File "./streams_err.py", line 5, in <module>
    #raise ValueError("Now we generate an error to STDERR")
#ValueError: Now we generate an error to STDERR
./streams_err.py < new_file.txt 2> error_file.txt
#This will come from STDIN: Now we write it to STDOUT: Don't mind #me,
just a bit of text here...
cat error_file.txt
#Traceback (most recent call last):
  #File "./streams_err.py", line 5, in <module>
    #raise ValueError("Now we generate an error to STDERR")
#ValueError: Now we generate an error to STDERR
echo "These are the contents of the file" > myamazingfile.txt
cat myamazingfile.txt
#These are the contents of the file

ls -l | less
#(... A list of files appears...)
cat spider.txt | tr ' ' '\n' | sort | uniq -c | sort -nr | head
```

```
# 7 the
# 3 up
# 3 spider
# 3 and
# 2 rain
# 2 itsy
# 2 climbed
# 2 came
# 2 bitsy
# 1 waterspout.
```

Managing files and directories

Many applications configure themselves by reading files. They are designed to read and write files in specific directories. Because of this, developers need to understand how to move and rename files, change their permissions, and do simple operations on their contents. Here are some common commands:

mv is used to move one or more files to a different directory, rename a file, or both at the same time.

Note: Linux is case-sensitive, so mv can also be used to change the case of a filename.

mv myfile.txt dir1/ This command moves a file to the directory.

mv file1.txt file2.txt file3.txt dir1/ This command moves multiple files.

cp is used to copy one or more files. Some examples include:

cp file1.txt file2.txt

cp file1.txt file2.txt file3.txt dir1/

chmod/chown/chgrp is used to make a file readable to everyone on the system before moving it to a public directory. A common example is:

chmod +r file.html && mv file.html /var/www/html/index.html

Operating with the content of files

Every programmer will use files for something. Whether it’s for configuration, data, or input and output, programmers work with files and need to know how to operate with their contents.

cut is a command that extracts fields from a data file. Two examples are:

cut -f1 -d”,” addressbook.csv This command extracts the first field from a .csv file.

cut -c1-3,5-7,9-12 phones.txt This command extracts only the digits from a list of phone numbers.

sort is a command that sorts the contents of a file. Some examples include:

sort names.txt This command sorts inputs alphabetically.

sort -r names.txt This command sorts inputs in reverse alphabetical order, starting with the letter z.

sort -n numbers.txt This command treats the inputs as numbers and then sorts them numerically.

Some examples that include combining multiple commands are:

ls -l | cut -w -f5,9 | sort -rn | head -10 This command displays the 10 largest files in the current directory.

cut -f1-2 -d”,” addressbook.csv | sort This command extracts the first and last names from a .csv file and sorts them.

Additional commands

Additional commands that programmers commonly use are:

id is a command that prints information about the current user. This command is useful if you are getting a permissions denied error and think you should be granted access to a file.

```
$ id
uid=3000(tradel) gid=3000(tradel)
groups=3000(tradel),0(root),100(users),545(builtin_users),999(docker)
```

free is a command that prints information about memory on the current system.

free -h This command prints in human-readable units instead of bytes.

Managing streams

These are the redirectors that we can use to take control of the streams of our programs

- command **>** file: redirects standard output, overwrites file
- command **>>** file: redirects standard output, appends to file
- command **<** file: redirects standard input from file
- command **2>** file: redirects standard error to file
- command1 **|** command2: connects the output of command1 to the input of command2

Operating with processes

These are some commands that are useful to know in Linux when interacting with processes. Not all of them are explained in videos, so feel free to investigate them on your own.

- **ps**: lists the processes executing in the current terminal for the current user
- **ps** ax: lists all processes currently executing for all users
- **ps** e: shows the environment for the processes listed
- **kill** PID: sends the SIGTERM signal to the process identified by PID
- **fg**: causes a job that was stopped or in the background to return to the foreground
- **bg**: causes a job that was stopped to go to the background
- **jobs**: lists the jobs currently running or stopped
- **top**: shows the processes currently using the most CPU time (press "q" to quit)

BASH

```
#!/bin/bash
echo "Starting at: $(date)"
echo

echo "UPTIME"
uptime
echo

echo "FREE"
free
echo

echo "WHO"
who
echo

echo "Finishing at: $(date)"
./gather-information.sh
```

About this code

Here, the starting and finishing times are the same, because there are so few operations we're doing that it takes the computer less than a second to complete them.

Code output:

```
Starting at: Mi 22. Mai 17:13:06 CEST 2019
UPTIME
 17:13:06 up 8 days,  1:34,  2 users,  load average: 0,00, 0,00, 0,00
FREE
              total        used        free      shared  buff/cache
available
Mem:        4037132      871336      253940       10032     2911856
2865984
Swap:       2097148        4364     2092784
WHO
user     :0           2019-05-14 15:39 (:0)
user     pts/1        2019-05-14 15:40 (192.168.122.1)
Finishing at: Mi 22. Mai 17:13:06 CEST 2019
```

```
#!/bin/bash

echo "Starting at: $(date)"; echo

echo "UPTIME"; uptime; echo

echo "FREE"; free; echo

echo "WHO"; who; echo

echo "Finishing at: $(date)"
./gather-information.sh
```

About this code

Here we can see the code is still working as expected!

Code output:

```
Starting at: Mon 13 May 2019 02:52:11 PM CEST
UPTIME
 14:52:11 up 17 days,  2:35,  1 user,  load average: 0.70, 1.01, 1.16
FREE
```

```
              total        used        free      shared  buff/cache
available
Mem:       32912600    19966400     1003304      321672    11942896
12281516
Swap:      20250620      612352    19638268
WHO
user    tty7         2019-04-29 12:19 (:0)
Finishing at: Mon 13 May 2019 02:52:11 PM CEST
```

```
#!/bin/bash

line="-------------------------------------------------"

echo "Starting at: $(date)"; echo $line

echo "UPTIME"; uptime; echo $line

echo "FREE"; free; echo $line

echo "WHO"; who; echo $line

echo "Finishing at: $(date)"
```

```
#!/bin/bash

n=1
while [ $n -le 5 ]; do
  echo "Iteration number $n"
  ((n+=1))
done
```

```
#!/bin/bash

n=0
command=$1
while ! $command && [ $n -le 5 ]; do
        sleep $n
        ((n+=1))
        echo "Retry #$n"
done;
```

```
#!/bin/bash

for file in *.HTM; do
        name=$(basename "$file" .HTM)
        echo mv "$file" "$name.html"
done
```

```
#!/bin/bash

for logfile in /var/log/*log; do
    echo "Processing: $logfile"
    cut -d' ' -f5- $logfile | sort | uniq -c | sort -nr | head -5
done
```

```
for i in $(cat story.txt); do B=`echo -n "${i:0:1}" | tr "[:lower:]" "[:upper:]"`; echo -n "${B}${i:1} "; done; echo -e "\n"
```

Terms and definitions from Course 2, Module 6

Bash script: A script that contains multiple commands

Cut: A command that can split and take only bits of each line using spaces

Globs: Characters that create list of files, like the star and question mark

Pipes: A process of connecting the output of one program to the input of another

Piping: A process of connecting multiple scripts, commands, or other programs together into a data processing pipeline

Redirection: A process of sending a stream to a different destination

Signals: Tokens delivered to running processes to indicate a desired action

IT skills in action reading

Congratulations! You have gained so much knowledge about using Python to interact with your operating system. There are many technical pieces that are included while using regexes in your code, but how would you apply the skills you learned in a professional setting?

In this reading, you will review an example of how regular expressions are used in the real world.

Disclaimer: The following scenario is based on a fictitious company called LogicLink Innovations.

Time is ticking

Dakota is a fairly new programmer with his company. He just earned a spot on the project for LogicLink Innovations. This is one of the biggest and most credible companies in the industry, so Dakota knows he has to excel on this project to help make a name for himself. LogicLink Innovations manages customer data and has hundreds of customer phone numbers in its database. The phone numbers are in inconsistent formats. Some are written with dashes, some in parentheses with spaces, and some are just digits. Dakota sees this:

```
123-456-7890
(123) 456-7890
1234567890
```

Dakota is assigned to take the dataset containing phone numbers and organize the formatting so they are all consistent. His manager tells him they need it by the end of the week! There is no way Dakota can work through and edit hundreds of phone numbers. There has to be another way.

Search and replace

Dakota remembers reading about how other programmers use regular expressions to make their coding life easier. He knows there has to be one that can help him with his dilemma. This can’t be the first time a programmer needs to standardize numbers! He decides to craft a regular expression that captures three groups of digits, each of which might be surrounded by non-digit characters. Using a regex tool and the sample data from above, he eventually comes up with a regex that matches all three samples:

```
^\D*(\d{3})\D*(\d{3})\D*(\d{4})$
```

Let’s break down this line of code, piece by piece:

`^\D*` **This part of the code matches zero or more non-digit characters at the beginning of the string.**

`(\d{3})` This part of the code captures exactly three digits, which represent the area code.

`\D*` This part of the code matches zero or more non-digit characters between the area code and exchange.

`(\d{3} )` This part of the code captures the three-digit exchange.

`\D*` This part of the code matches zero or more non-digit characters between the exchange and line.

`(\d{4})$` This part of the code captures exactly four digits at the end of the string.

Now he has three capture groups: area code, exchange, and number. He then substitutes those groups into a new string using backreferences:

```
(\1) \2-\3
```

This puts all of the phone numbers into a uniform format.

This regular expression helps Dakota by searching for phone numbers in different formats and replacing them to match the format that Dakota’s manager needs: (123) 456-7890. Dakota begins to code.

He writes up a simple Python script to read the dataset from a file and output the corrected phone numbers using his regular expressions:

```
import re
with open("data/phones.csv", "r") as phones:
 for phone in phones:
   new_phone = re.sub(r"^\D*(\d{3})\D*(\d{3})\D*(\d{4})$", r"(\1) \2-\3", phone)
   print(new_phone)
(123) 456-7890
(123) 456-7890
(123) 456-7890
```

Steps for Coding Projects

1. Understand the problem statement
    a. What needs to be done
    b. Identify inputs / outputs
2. Research
    a. How to tackle problem using external modules
        i. Look at documentation with examples for modules and func
    b. Others have solved something similar before
3. Planning
    a. What data types
    b. Order of operations
    c. Writing down the plan in design document
4. Writing
    a. Also test the code
        i. Manually and with automation
    b.