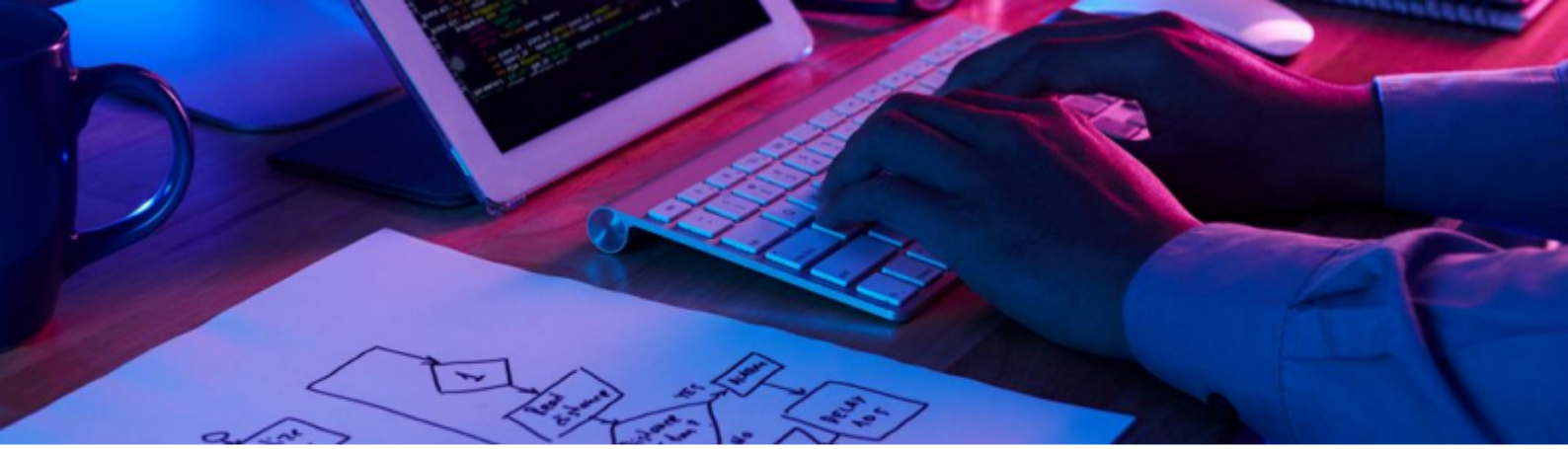


Using Python to Interact with the Operating System

Course 2 of 6 in Google IT Automation with Python Specialization.

bangkit

Running Python on Local & Remote Computer

Getting Ready for Python

- To check whether you already have Python installed, open a terminal or command prompt and execute Python command, passing --version as a parameter.
- Result similar to "unrecognized command" means no Python installed. Follow instruction on the course to install on your computer.

Check on terminal (OS Linux or MacOS)

shell prompt, no need to type $

```
$ python --version
Python 3.7.9

$ python3 --version
Python 3.7.9
```

as alternative, also check python3

Python installed, version 3.7.9

Check on Command Prompt or PowerShell (OS Windows)

command prompt, no need to retype

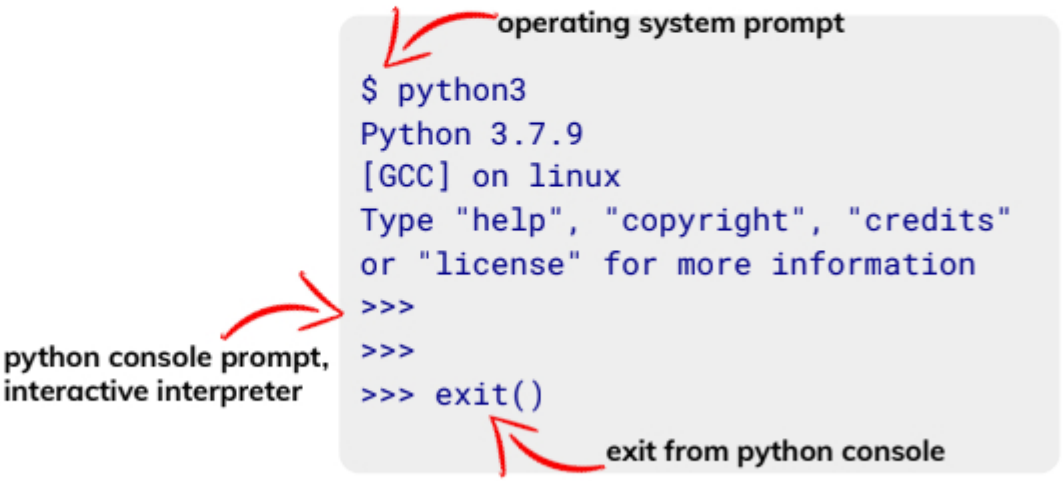
```
C:\Users\bangkit> python
--version
Python 3.7.9
```

Python installed, version 3.7.9

Getting Ready for Python

- Run python console

Run command python3
(to enter the python console)



Getting Ready for Python

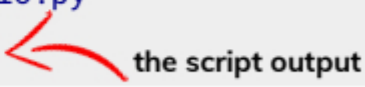
- Write text file, with text editor, save file as hello.py (extension .py for python script).
- Run a Python script.

Content of file hello.py

```
print("Hello, World")
```

Run Python Script hello.py

```
$ python3 hello.py
Hello, World
```



Getting Ready for Python

- Python modules is a directory that contains special file __init__.py (double-underscore init file). Python modules also called packages.
- The **`__init__.py`** is mostly an empty file, but also available to setup file imports, will explore more on later topic.
- Command **`ls`** in Linux/MacOS means list, equal to command **`dir`** in Windows.

Listing Module Directory

```
$ ls mymodule/
__init__.py    hello.py
```

Getting Ready for Python

- Working with Text Editor or IDE.
  - To edit the python script code, go with whatever installed in computer like nano, vim, emacs on Linux, Notepad, TextEditor on Windows, TextEdit on MacOS.
  - More advanced code editor also available for free like Atom, Notepad++, or Sublime Text.
  - Or use a fully featured graphical IDE like PyCharm, Eclipse, or Visual Studio Code.

Getting Ready for Python

- Automating tasks through programming.
  - Automation is an incredibly powerful tool that saves time, reduces mistakes, and facilitates growth and scalability.
  - Concept called the Pareto Principle presents itself as a useful guideline to decide which tasks to automate. Pareto Principle states that 20% of system admin tasks are responsible for 80% of work. Automate those 20% tasks to save a whole lot of time.
  - Need to apply automation thoughtfully to avoid some pitfalls that may arise from its use.

Getting Ready for Python

- Graded assessment using Qwiklabs online platform.
  - Qwiklabs works with the Google Cloud Console to spin up and create **virtual machines**, (VM for short), a computer simulation through software.
  - During the lab, access instances of Qwiklabs VMs by using SSH, a command that allows you to interact with **remote Linux computers**. Once it's completed, Qwiklabs destroy all the VMs.
  - Get experience solving real-world tasks through scripting by using Qwiklabs.

Python on Linux Operating System

Managing Files with Python

- Reading files using Python.
- Function **open** will start to open the file , if has read access, and return a file descriptor to do more operation with the file.
- Function **readline** will read single line from file. Function **read** will read all the way through to the files end.
- Don't forget to call function **close** to release the file descriptor and finalize the process on the file.

Read Existing spider.txt File.

python console prompt

```
>>> file = open("spider.txt")
>>> print(file.readline())
The itsy bitsy spider climbed up
the waterspout.
>>> print(file.read())
Down came the rain
and washed the spider out.
>>> file.close()
```

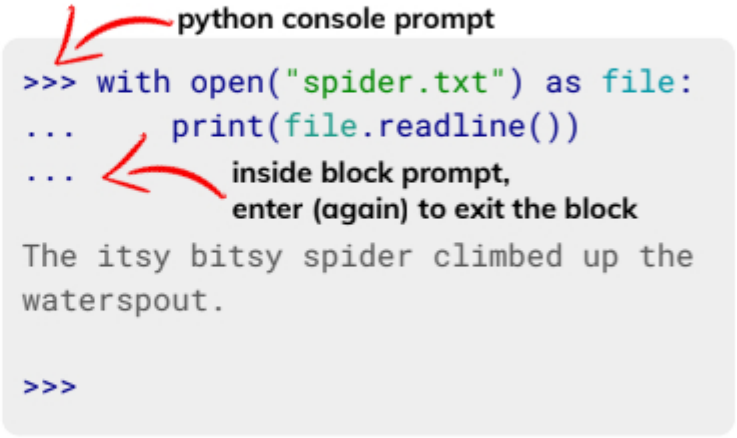
Content of File spider.txt

```
The itsy bitsy spider climbed up the
waterspout.
Down came the rain
and washed the spider out.
```

Managing Files with Python

- Read files using Python.
- To make sure that all opened file always closed, use an alternative to write it as a block of code by using keyword with. When the program going out from the with block, python will automatically close the file.

Read Existing spider.txt File.



Managing Files with Python

- Iterating through files using Python.
- Using function **readlines** to read the file contents as list of string, splitted line by line. The new list still can be read even the file closed.
- Function readlines (also readline), is reading file by line yet keeping the new line (\n) character at the end of the read string.
- Character with backslash like new line (\n) is called escape character.

Read Existing spider.txt File.

```
>>> with open("spider.txt") as file:
...     lines = file.readlines()
...

>>> lines.sort()  # "sort alphabetically"
>>> print(lines)
['Down came the rain\n', 'The itsy
bitsy spider climbed up the
waterspout.\n', 'and washed the
spider out.\n']
```

Managing Files with Python

- Writing files using Python.
- Using the same with block, function open by default using mode "r" (read-only), can be used to write file with mode "w" (write-only), if file exists will be overwritten. To avoid, use mode "a" for append, or mode "r+" for read-write.
- Function **write** returns the number of characters that are successfully written.

Read Existing spider.txt File.

```
>>> mode = "w"
>>> with open("novel.txt", mode) as file:
...     file.write("It was a dark and
stormy night")
...
30
>>>
```

* Complete modes for function open in Python docs
https://docs.python.org/3/library/functions.html#open

Managing Files with Python

- Module **os** provides a layer of abstraction between Python and the operating system.
- To check if a file exists, use function **exists** from sub-module **path**.
- Rename a file with function **rename** and delete a file using function **remove**.

Working with Files.

```
>>> import os
>>> os.path.exists("novel.txt")
True
>>> os.rename("novel.txt", "new.txt")
>>> os.path.exists("novel.txt")
False
>>> os.remove("new.txt")
```

Managing Files with Python

- Work with files and directories.
- Get current working directory (cwd) with function **getcwd**. To change it, use function **chdir**.
- Make a new directory use **mkdir**, remove (an empty) directory with function **rmdir**.
- Find out what's inside directory using **listdir**, also check if it is a directory with function **isdir** or a file with **isfile**.

Working with Directories.

```
>>> import os
>>> print(os.getcwd())
/home/bangkit
>>> os.mkdir("new_dir")
>>> os.chdir("new_dir")
>>> print(os.getcwd())
/home/bangkit/new_dir
>>> os.chdir("..")  # up 1 dir
>>> os.rmdir("new_dir")
>>> os.listdir(".")  # current dir
['data', 'spider.txt']
>>> os.path.isdir("data")
True
```

Managing Files with Python

- CSV stands for Comma Separated Values is a spreadsheet-like file, where each line corresponds to a row and each comma separated field corresponds to a column.
- Use **reader** function from **csv** module to read CSV from opened file descriptor.

Reading CSV Files

```
>>> import csv
>>> f = open("data.csv")
>>> csv_f = csv.reader(f)
>>> for row in csv_f:
...     name, phone, role = row
...     print(name+': '+role)
...
Sabrina Green: System Administrator
Eli Jones: IT specialist
>>> f.close()
```

Content of File data.csv

```
Sabrina Green,802-867,System Administrator
Eli Jones,684-348,IT specialist
```

Managing Files with Python

- Writing CSV files, from list of list:
  - Main list as the rows
  - Detail list (inside) as columns
- Module csv has function **writerow** to write single row and **writerows** to write multiple rows at once.

Generating CSV

```
>>> hosts = [
    ["ws.local", "192.168.1.2"],
    ["web.cloud", "192.168.1.1"],
]
>>> with open('hosts.csv', 'w') as h_csv:
...     writer = csv.writer(h_csv)
...     writer.writerows(hosts)
...
>>>
```

Regular Expressions

- Dot (.) which is matched to any character, in the regex world is known as wildcard because it can absolutely match any character.
- To restrict our wildcards to a range of characters, use another feature of regexes called character classes that are written inside square brackets. Use circumflex (^) inside square brackets to match any characters that **are not in a group**.

Wildcards and Character Classes

```
>>> print(re.search(r"[Pp]ython", "Python"))
<re.Match object; span=(0, 6), match='Python'>
>>> print(re.search(r"[a-z]way", "the highway"))
<re.Match object; span=(7, 11), match='hway'>
>>> print(re.search(r"[a-z]way", "a way to go"))
None
>>> print(re.search(r"cloud[a-zA-Z0-9]", "cloudy"))
<re.Match object; span=(0, 6), match='cloudy'>
>>> print(re.search(r"cloud[a-zA-Z0-9]", "cloud9"))
<re.Match object; span=(0, 6), match='cloud9'>

>>> print(re.search(r"[^a-zA-Z]", "This is a
sentence."))
<re.Match object; span=(4, 5), match=' '>

>>> print(re.search(r"[^a-zA-Z ]", "This is a
sentence."))
<re.Match object; span=(18, 19), match='.'>
```

Regular Expressions

- Use the pipe symbol (|) to match one expression or another. Function **search** only match the first pattern which it finds.
- To get all possible matches, use the **findall** function, also from **re** module.

Match One or More Expressions

```
>>> print(re.search(r"cat|dog", "I like cats."))
<re.Match object; span=(7, 10), match='cat'>
>>> print(re.search(r"cat|dog", "I like dogs."))
<re.Match object; span=(7, 10), match='dog'>
>>> print(re.search(r"cat|dog", "I like both
dogs and cats."))
<re.Match object; span=(12, 15), match='dog'>

>>> print(re.findall(r"cat|dog", "I like both
dogs and cats."))
['dog', 'cat']
>>>
```

Regular Expressions

- Repeated matches is another regex concept. For example, using a **dot** followed by a **star**, means that it matches any character repeated as many times possible **including zero**. The **star** takes as many character as possible, called greedy in programming terms.
- The **plus** character matches one or more occurrences of the character before it.
- The **question mark** symbol means either zero or one occurrence of the character before it.

```
>>> print(re.search(r"Py.*n", "Pygmalion"))
<re.Match object; span=(0, 9), match='Pygmalion'>
>>> print(re.search(r"Py.*n", "Python Programming"))
<re.Match object; span=(0, 17), match='Python Programmin'>
>>> print(re.search(r"Py[a-z]*n", "Python
Programming"))
<re.Match object; span=(0, 6), match='Python'>
>>> print(re.search(r"Py[a-z]*n", "Pyn"))  # 0
<re.Match object; span=(0, 3), match='Pyn'>

>>> print(re.search(r"o+l+", "goldfish"))
<re.Match object; span=(1, 3), match='ol'>
>>> print(re.search(r"o+l+", "woolly"))
<re.Match object; span=(1, 5), match='ooll'>
>>> print(re.search(r"o+l+", "boil"))
None

>>> print(re.search(r"p?each", "To each their own"))
<re.Match object; span=(3, 7), match='each'>
>>> print(re.search(r"p?each", "I like peaches"))
<re.Match object; span=(7, 12), match='peach'>
```

Regular Expressions

- To match special characters like dot (.), star (*), plus (+), question mark (?), circumflex (^), dollar sign ($), square brackets ([]), use escape characters.
- Python also uses the backslash for a few special sequences that used to represent predefined characters. Eg: \w matches any alphanumeric character including letters, numbers, and underscores.

Escaping Characters

```
>>> print(re.search(r".com", "welcome"))
<re.Match object; span=(2, 6), match='lcom'>
>>> print(re.search(r"\.com", "welcome"))
None
>>> print(re.search(r"\.com", "google.com"))
<re.Match object; span=(6, 10), match='.com'>

>>> print(re.search(r"\w*", "This is test"))
<re.Match object; span=(0, 4), match='This'>
>>> print(re.search(r"\w*", "And_this_one"))
<re.Match object; span=(0, 12),
match='And_this_one'>
```

* More about regex introduction visit https://www.regex101.com

Regular Expressions

- Circumflex (^) pattern matches the beginning of the line. Dollar sign ($) pattern match the end of the line.
- Build pattern of valid variable name of Python:
  - It can contain any number of letters, numbers, or underscores
  - Can't start with a number

Regular Expressions in Action

```
>>> print(re.search(r"A.*a", "Argentina"))
<re.Match object; span=(0, 9), match='Argentina'>
>>> print(re.search(r"A.*a", "Azerbaijan"))
<re.Match object; span=(0, 9), match='Azerbaija'>
>>> print(re.search(r"^A.*a$", "Azerbaijan"))
None
>>> print(re.search(r"^A.*a$", "Australia"))
<re.Match object; span=(0, 9), match='Australia'>

>>> pattern = r"^[a-zA-Z_][a-zA-Z0-9_]*$"
>>> print(re.search(pattern, "_variable_name"))
<re.Match object; span=(0, 14),
match='_variable_name'>
>>> print(re.search(pattern, "this isn't valid"))
None
>>> print(re.search(pattern, "my_variable1"))
<re.Match object; span=(0, 12),
match='my_variable1'>
>>> print(re.search(pattern, "2my_variable1"))
None
```

Managing Data and Processes

- Standard streams: standard input, standard output, standard error.

streams.py

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

data = input("from STDIN: ")
print("write to STDOUT: "+data)
print("error to STDERR: "+data+1)
```

Use streams.py as executable

```
$ chmod u+x streams.py
$ ./streams.py
from STDIN: BANGKIT!
write to STDOUT: BANGKIT!
TypeError: can only concatenate
str (not "int") to str
```

Managing Data and Processes

- When open a terminal application on Linux, the application that reads and executes all commands is called a shell, a command line interface to interact with operating system.
- This course applies **bash** as the Linux shell. Use **env** command to show all environment variables on **bash** shell.
- **echo** command will print an environment variable, add $ sign prefix before the variable name.

Using bash shell

```
$ env
... (many outputs)
$ echo $PATH
/usr/bin:/bin:/usr/games
$ echo $HOME
/home/bangkit
```

Get Data From Environment Variable

```
#!/usr/bin/env python3
import os
print("HOME: "+os.environ.get("HOME", ""))
print("SHELL: "+os.environ.get("SHELL", ""))
```

Managing Data and Processes

- Access value of command line arguments with the argv in the **sys** module.

parameters.py

```
#!/usr/bin/env python3
import sys
print(sys.argv)
```

Execute app with Arguments

```
$ ./parameters.py
['./parameters.py']

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

Managing Data and Processes

- Exit status is a value returned by a program to the shell. In all Unix-like operating systems, the exit status of the process is zero when the process succeeds and different than zero if it fails.
- To get the exit status from previously run command, print the contents of the $? variable (dollar sign question mark).

create_file.py

```
#!/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\n")
else:
    print("Error, {} exists"
          .format(filename))
    sys.exit(1)
```

Execute app with Arguments

```
$ ./create_file.py example
$ echo $?
0
$ ./create_file.py example
Error, example exists
$ echo $?
1
```

Managing Data and Processes

- Use system command as part of Python script to accomplish a task, with functions provided by the subprocess module.
- To run external command the script create subprocess, then it's blocked while waiting for the subprocess to finish.

Running System Command

```
>>> import subprocess
>>> subprocess.run(["date"])
Tue 01 Dec 2020 10:11:12 AM WIT
CompletedProcess(args=['date'], returncode=0)

>>> subprocess.run(["sleep", "7"])
CompletedProcess(args=['sleep', '7'],
returncode=0)
>>>
```

Managing Data and Processes

- To get the command output, set argument capture_output in function run to True. This argument available only for Python 3.7 or newer.
- The command output will be accessible through attribute stdout for standard output and stderr for standard error.

Running System Command

```
>>> from subprocess import run
>>> result = run(["host", "8.8.8.8"],
                 capture_output=True)
>>> print(result.returncode)
0
>>> print(result.stdout)
b'8.8.8.8.in-addr.arpa domain name pointer
dns.google\n'

>>> result = run(["rm", "not_a_file"],
                 capture_output=True)
>>> print(result.returncode)
1
>>> print(result.stderr)
b"rm: cannot remove 'not_a_file': No such
file or directory\n"
```

Managing Data and Processes

- Many events that are taking place in programs running in a system and aren't connected to terminal are usually write log files.
- Create Python program to read the log line by line instead of loading the entire contents info memory.
- After getting the log line, explore how to get the user name which is triggered the CRON jobs (on next slide).

Sample System Log (syslog)

```
Jul 6 14:01:23 host.name CRON[294]: USER (good_user)
Jul 6 14:02:08 host.name jam_tag=psim[291]: (UUID:006)
Jul 6 14:03:01 host.name CRON[294]: USER (naughty_user)
Jul 6 14:04:01 host.name CRON[294]: USER (naughty_user)
```

Find CRON Jobs Log

```
#!/usr/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())
```

Managing Data and Processes

- The username is found **at the end** of line, use the $ sign to match only end of line.
- Find the username by looking for the **word USER** followed by a string wrapped in parentheses. Escape the parentheses with a backslash.
- Extract the actual username with another couple of parentheses to create a **capturing group**. Matching the username with any **alphanumeric** characters by using backslash w plus.

Filter Log with Regular Expressions

```
>>> import re
>>> pattern = r"USER \((\w+)\)$"
>>> line = "Jul 6 14:04:01 host.name
CRON[294]: USER (naughty_user)"
>>> result = re.search(pattern, line)
>>> print(result[1])
naughty_user
>>>
```

Testing in Python

- Software testing is a process of evaluating computer code to determine whether or not it does what it is expected to do.
- Use the interpreter to try our code before putting it in a script is another form of **manual testing**. Formal software testing, called **automatic testing**, codifying tests into its own software and code that can be run to verify that the programs do what expected to do.
- The **goal of automatic testing** is to automate the process of checking if the returned value matches the expectations.

Testing in Python

- Unit tests are used to verify that small isolated parts of a program are correct.
- The convention in Python is to call the script with the same name of the module that it's testing and appending the suffix `_test`.
- Use module **unittest** to create a unit test. Use the **assertEqual** method to verify that the expectation is exactly the same as to what is returned by the tested function.

bangbit

Testing in Python

rearrange_test.py

```
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)

unittest.main()
```

Function to run the test

rearrange.py

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

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

Testing in Python

- Add more test cases.
- Add edge cases which will input to our code so it produces unexpected results. Edge case is often found at the extreme ends of the ranges of input imagined that the programs will typically work.
- Update the main program to resolve the test cases.

rearrange_test.py

```
from rearrange import rearrange_name
import unittest

class TestRearrange(unittest.TestCase):
    def test_basic(self): pass  # use above
    def test_empty(self):
        testcase = ""
        expected = ""
        self.assertEqual(
            rearrange_name(testcase),
            expected)

unittest.main()
```

Run the test

```
$ python rearrange_test.py
..
----------------------
Ran 2 tests in 0.000s
OK
```

Testing in Python

- **White-box** testing (AKA clear-box or transparent testing) relies on the test creator's knowledge of the tested software to construct the test cases. On the other hand in **Black-box** testing, the tested software is treated like an opaque box. The tester doesn't the internals of how the software works.
- If the unit tests are created **before** any code is written based on specifications of what the code is supposed to do, it can be considered as a **black-box** unit test. If unit tests are run **alongside** or **after** the code has been developed, the test cases are made with a knowledge of how software works, is considered as a **white-box** tests.
- Process called **test-driven development** or **TDD** calls for creating the test before writing the code.

Testing in Python

- In most cases, error generated by Python can be handled with try-except construct. It means that the program acknowledges the (expected) error and performs an alternate execution.

Try-Except Construct.

```
def get_spider_text():
    try:
        f = open("spider.txt")
        lines = f.readlines()
        f.close()
    except OSError:
        lines = None

    return lines
```

this line executed only if error opening file

Testing in Python

- In some other cases, the program is not exclusive to handle error, but also raise error.
- For example, inside a function, if the input parameters not as expected as on the specification, instead of return unknown value, it would conventionally be better to raise error.

Raising Errors

```
def validate_user(username, minlen):
    if minlen < 1:
        raise ValueError("minlen at least 1")
    if len(username) < minlen:
        return False
    if not username.isalnum():
        return False
    return True
```

Testing in Python

- To test expected errors on **unittest**, use assertRaises method.

Testing for Expected Errors

```
from validations import validate_user
import unittest

class TestValidateUser(unittest.TestCase):
    # some other tests

    def test_invalid_minlen(self):
        self.assertRaises(
            ValueError,
            validate_user, "user", -1)

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

Bash Scripting

- Up to this course, many Linux commands has been used. Here's some of them:
  - **echo**: print information (like environment variable) to standard output
  - **cat** file: shows the content of the file through standard output
  - **ls**: lists the contents of the current directory
  - **cd** directory: change current working directory to the specified one
  - **rm**: remove file or directory (with specific arguments)
  - **chmod** modifiers files: change permissions for the files according to the provided modifiers
- On Unix-based system, the command documentation can usually be found in manual or man pages, using **man** command.

Bash Scripting

- Redirection is a process of sending a stream to a different destination.
- To redirect the standard output of a program, use the greater than symbol (>) to overwrite the file. To append, use the double greater than symbol (>>).
- To redirect standard input, instead of using keyboard to send data to a program, use the less than symbol (<) to read the contents of a file.
- To redirect standard error, use character combination 2>

Bash Scripting

Stream Redirections

```
$ cat stdout_example.py
#!/usr/bin/env python3
print("a bit of text.")
$ ./stdout_example.py > new_file.txt
$ cat new_file.txt
a bit of text.
$ ./stdout_example.py >> new_file.txt
$ cat new_file.txt
a bit of text.
a bit of text.

$ ./streams.py < new_file.txt 2> err.txt
from STDIN: a bit of text
$ cat err.txt
TypeError: can only concatenate str (not "int") to str
```

* streams.py from managing data and processes.

Bash Scripting

- Pipes connect the output of one program to the input of another in order to pass data between programs. Pipes are represented by the pipe character (|).
- Pipes will create new commands by combining the functionality of one command, with the functionality of another, without having to store the contents in intermediate file.

Pipes and Pipelines

```
$ cat capitalize.py
#!/usr/bin/env python3
import sys
for line in sys.stdin:
    print(line.strip().capitalize())
$ cat spider.txt | ./capitalize.py
The itsy bitsy spider climbed up the waterspout.
Down came the rain
And washed the spider out.
```

* spider.txt from managing files with python.

Bash Scripting

- Another way of communication between process in Linux is through the use of signals. Signals are tokens delivered to running process to indicate a desired action.
- Eg: interrupting ping command with **Ctrl-C** will send the **SIGINT**. Or send **SIGSTOP** to a running command with **Ctrl-Z**, continue it later with command **fg** (foreground).
- To send other signals use command **kill**, by default send **SIGTERM** to terminate process. Command kill needs the process id (PID) of the running command, use command **ps**, or with arguments: **ps ax**.

Bash Scripting

- Bash is not only the interpreter that runs commands, it's also a scripting language. Use bash to write simple scripts to use a lot of commands.
- Write the script by one command per line or multiple commands on the same line using semicolons to separate them.

Create Bash Scripts info.sh

```
#!/bin/bash
dt=$(date)
echo "Starting at ${dt}"
echo "UPTIME"
uptime
echo
echo "WHO"; who; echo
```

run command then save output to variable. using parentheses.

combine text & variable. using curly braces.

one command per line

multiple commands using semicolons

Execute Bash Scripts

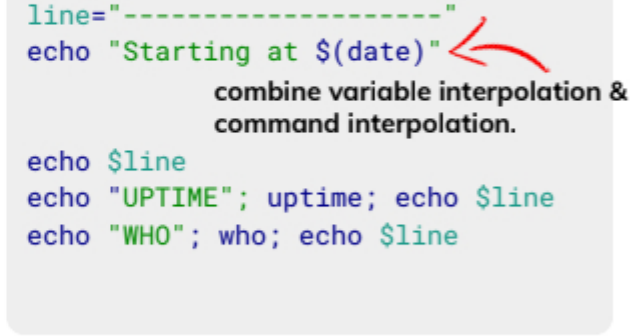
```
$ chmod u+x info.sh
$ ./info.sh
Starting at: Thu 03 Dec 2020 10:11:12 AM WIT
UPTIME
 10:11:12 up 2 days, 20:30, 1 user
WHO
User    tty7    2020-12-03 10:11 (:0)
```

Bash Scripting

- To access the value of a variable in bash, prefix the variable name with the dollar sign ($). Like previously used to access environment variables.
- To define a variable use equal sign (=) with no spaces between the variable name, equal sign and the value.
- Use **export** command to export variable between environment, or else it only available as local variable.

Bash Scripts info.sh

```
#!/bin/bash
line="--------------------"
echo "Starting at $(date)"
echo $line
echo "UPTIME"; uptime; echo $line
echo "WHO"; who; echo $line
```



Bash Scripting

- In bash scripting, an exit value of 0 means success (true condition). Indentation is not mandatory in bash.
- test is a command that evaluates the conditions received and exits with zero when it's true and with one if false. Square bracket ([]) is an alias to test command, the syntax needs to be a space after/before (inside) the brackets.

Conditional Execution in Bash

```
#!/bin/bash
if grep "127.0.0.1" /etc/hosts; then
    echo "localhost found"
else
    echo "ERROR localhost not found"
fi
```

Evaluate Condition with test

```
$ if test -n "$PATH"; then echo "Path not
empty"; fi
Path not empty

$ if [ -n "$PATH" ]; then echo "Path not
empty"; fi
Path not empty
```

Bash Scripting

- The condition for the while loop uses the same format as a condition for an if block.
- To increment the value of the variable n, using a bash construct of double parentheses, which is do an arithmetic operations.

While Loops in Bash Scripts

```
#!/bin/bash
command=$1

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

similar to sys.argv in Python

keep trying if command NOT success (exit 0)

test if less than or equal to 5

Bash or Python

- It's a good idea to choose bash when operating with files and system commands, as long as it is simple enough that the script is self-explanatory.
- As soon as it gets harder to understand what the script is doing, it's better to write it in a more general scripting language like Python.
- If the tasks are limited to the current server or a fleet of servers with the same operating system, bash script can get the job done. But if it is too complex or it needs to work across platform, it might be better off using Python.


bangkit