GraphCore Requests

1. Spawning other processes (subprocess)
2. Regular Expressions
3. Command Line Tools

Spawning Other Processes

- Interacting with the OS
-

Interacting with the OS

There are three key modules that allow us to interact with the host operating system

- The os module enables miscellaneous interfaces with the OS. You can use it to inspect environment variables or to get other user and process-related information.
- The platform module contains information about the interpreter and the machine where the process is running
- The sys module, which provides you with helpful system-specific information, will usually provide you with all the information that you need about the runtime environment.

The subprocess module

- Allows us to start a new process and communicate with it
- This module has gone through some work to modernize and simplify its API and you might see code using subprocess in ways different from those shown here
- There are two main APIs: the `subprocess.run` which is quite high-level and `subprocess.Popen` which is more low level
- We'll focus on the higher level API.
- Dig into the docs if you need more complex interactions

Using subprocess

```
1   import subprocess
2   subprocess.run(["ls"])
```

This will run the command in your shell. If you don’t have the command you’ll raise an exception.

Capture the output

```
import subprocess
result = subprocess.run(["ls"], capture_output=True)
print("stdout: ", result.stdout)
print("stderr: ", result.stderr)

```

Adding `text=True` will allow the `stdout` and `stderr` to be strings we can work with.

```
result = subprocess.run(["ls"], capture_output=True, text=True)
```

Pass Parameters

```
1   import subprocess
2   result = subprocess.run(["ls", "-l"], capture_output=True, text=True)
3   print("stdout: ", result.stdout)
4   print("stderr: ", result.stderr)
```

Add flags and argumenet in the list. These will be passed through to the shell in the order they are passed.

Checking the return code

```
1   result = subprocess.run(["ls", "non_existing_file"])
2   print("rc: ", result.returncode)
```

You can use the returncode attribute to decide if the process ended without error.

- 0 is success
- Anything else is some kind of failure

Running another Python script

- You can use `sys.executable` to use the Python executable that is currently running
- We can use this to then run another script

```
import subprocess
import sys

subprocess.run([sys.executable, "subprocess2.py"])

```

Regular Expressions

- Overview
- Basic Syntax
- Usage

Overview

- A domain specific language which defines a grammar for expressing string comparisons
- Easier than having loops of checking for different things
- It’s a standard language used across programming languages
- Python’s implementation is from the `re` package in the standard library.
- Built into lots of text editors (including PyCharm and VSCode)

Basic Syntax

- Most characters match their own identities, so "h" in a regex means "match exactly the letter h."
- Enclosing characters in square brackets can mean choosing between alternates, so if we thought a web link might be capitalized, we could start with "[Hh]" to mean "match either H or h." In the body of the URL, we want to match against any non-whitespace characters, and rather than write them all out. We use the \S character class. Other character classes include \w (word characters), \W (non-word characters), and \d (digits).
- Two quantifiers are used: ? means "0 or 1 time," so "s?" means "match if the text does not have s at this point or has it exactly once." The quantifier, +, means "1 or more times," so "\S+" says "one or more non-whitespace characters." There is also a quantifier *, meaning "0 or more times." Additional regex features that you will use in this chapter are listed here:
- Parentheses () introduce a numbered sub-expression, sometimes called a "capture group." They are numbered from 1, in the order that they appear in the expression.
- A backslash followed by a number refers to a numbered sub-expression, described previously. As an example, \1 refers to the first sub-expression. These can be used when replacing text that matches the

Usage

```
import re

title = "And now for something completely different"
pattern = "(\w)\\1+"

print(re.search(pattern, title))
print(re.findall(pattern, title))

reObj = re.compile(pattern)

print(reObj.search(title))
```

Another example

```
import re

description = "The Norwegian Blue is a wonderful parrot. This parrot is notable for its exquisite plumage."

pattern = "(parrot)"
replacement = "ex-\\1"

print(re.sub(pattern, replacement, description))

```

Command Line Tools

- Parse the arguments
- Carry out some work

Parse the arguments

There are a number of ways to parse the arguments in Python scripts. A reliable way to do this is with `argparse`.

First, we'll import the library:

```
import argparse
```

We create a parser object:

```
# create a parser object
parser = argparse.ArgumentParser(description="A description for our program")
```

Add any arguments we'd like to:

```
parser.add_argument("-a", "--add", nargs='*', metavar="num", type=int,
                    help="All the numbers separated by spaces will be added.")

```

Further Resources

- Mastering Python Regular Expressions book - https://www.packtpub.com/product/mastering-python-regular-expressions/9781783283156
- Automate the Boring Stuff with Python - https://automatetheboringstuff.com/