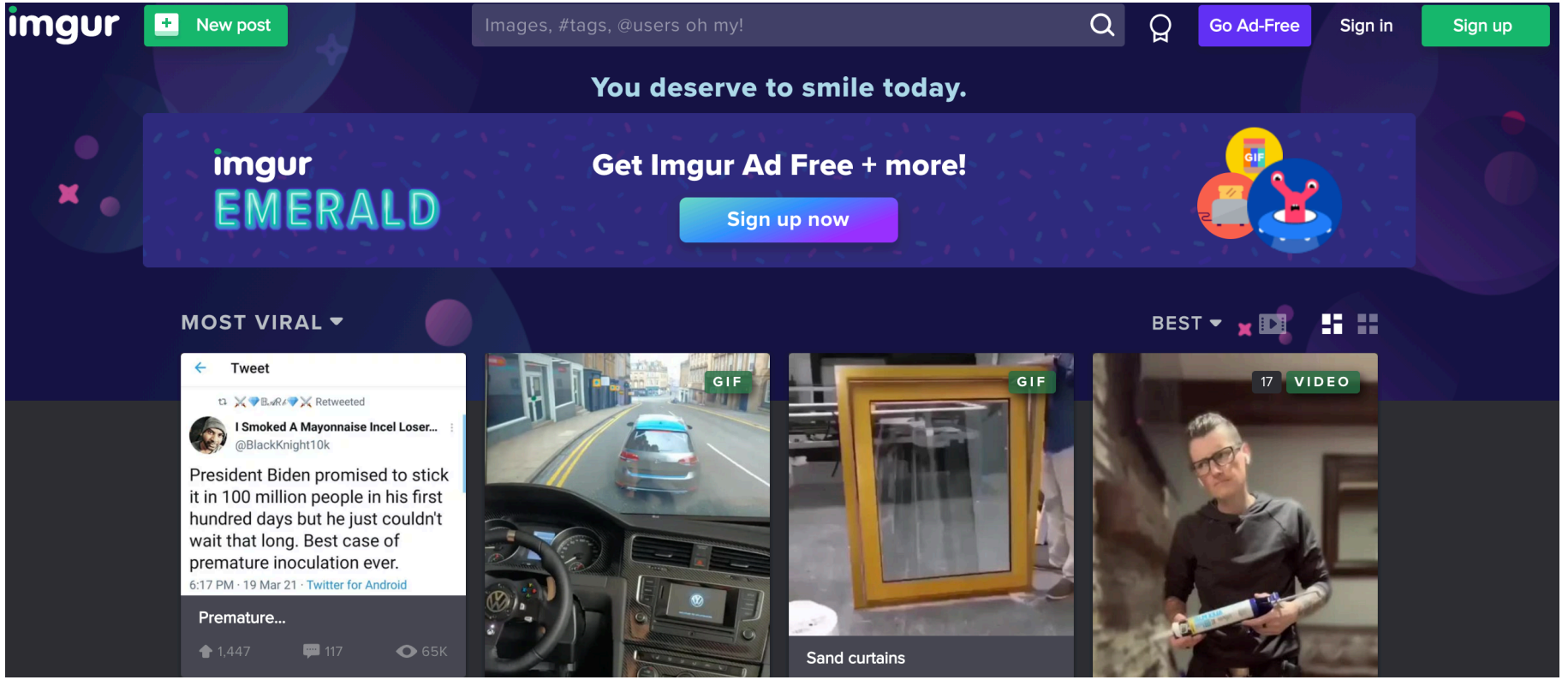
Python Concurrency

Getting Started

Ackwonledgement: https://nyu-cds.github.io/python-concurrency/

Task: fetch images from website: Imgur



Task: fetch images from website: Imgur

- Imgur provides a web API that allows images to be downloaded.
  - https://apidocs.imgur.com/#intro
- You will need a login and to register an application to use Imgur.
- Create a Python module, named download.py
  - https://github.com/Imgur/imgurpython
- Contains three separate functions:
  - get_links
  - download_link
  - setup_download_dir

Task: fetch images from Imgur

```
import json
import os
from pathlib import Path
from urllib.request import urlopen, Request

types = {'image/jpeg', 'image/png', 'image/gif'}

def get_links(client_id):
    headers = {'Authorization': 'Client-ID {}'.format(client_id)}
    req = Request('https://api.imgur.com/3/gallery/random/random/', headers=headers, method='GET')
    with urlopen(req) as resp:
        data = json.loads(resp.read().decode('utf-8'))
    return [item['link'] for item in data['data'] if 'type' in item and item['type'] in types]

def download_link(directory, link):
    download_path = directory / os.path.basename(link)
    with urlopen(link) as image, download_path.open('wb') as f:
        f.write(image.read())

def setup_download_dir():
    download_dir = Path('images')
    if not download_dir.exists():
        download_dir.mkdir()
    return download_dir
```

Task: fetch image

```
import json
```

JSON (**JavaScript Object Notation**, pronounced /ˈdʒeɪsən/; also /ˈdʒeɪˌsɒn/) is an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). It is a very common data format, with a diverse range of applications, such as serving as a replacement for XML in AJAX systems.[1]

Syntax [edit]

The following example shows a possible JSON representation describing a person.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Task: fetch images from Imgur

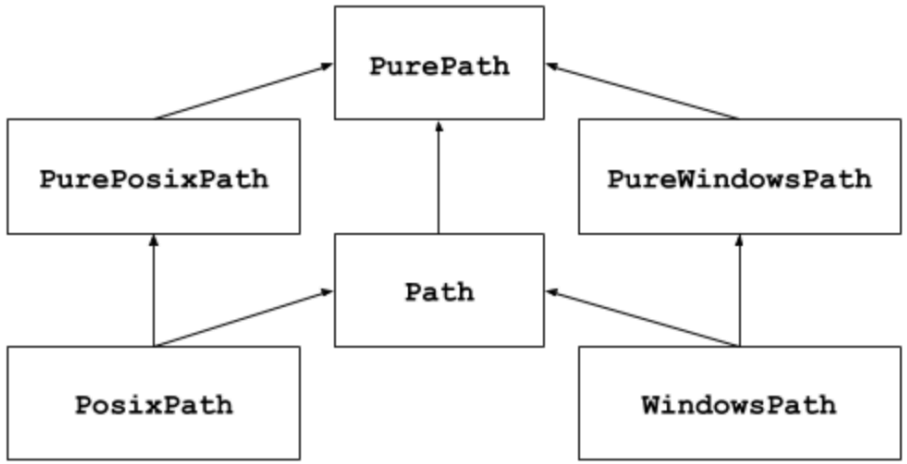
```
import json
import os
from pathlib import Path
```

`pathlib` — Object-oriented filesystem paths

New in version 3.4.

Source code: Lib/pathlib.py

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between pure paths, which provide purely computational operations without I/O, and concrete paths, which inherit from pure paths but also provide I/O operations.



Task: fetch images from Imgur

```
import json
import os
from pathlib import Path
from urllib.request import urlopen, Request
```

`urllib` — URL handling modules

Source code: Lib/urllib/

`urllib` is a package that collects several modules for working with URLs:

- `urllib.request` for opening and reading URLs
- `urllib.error` containing the exceptions raised by `urllib.request`
- `urllib.parse` for parsing URLs
- `urllib.robotparser` for parsing `robots.txt` files

Task: fetch images from Imgur

```
import json
import os
from pathlib import Path
from urllib.request import urlopen, Request
```

`urllib.request` — Extensible library for opening URLs

Source code: Lib/urllib/request.py

The `urllib.request` module defines functions and classes which help in opening URLs (mostly HTTP) in a complex world — basic and digest authentication, redirections, cookies and more.

Task: fetch images from Imgur

```
import json
import os
from pathlib import Path
from urllib.request import urlopen, Request

types = {'image/jpeg', 'image/png', 'image/gif'}

def get_links(client_id):
    headers = {'Authorization': 'Client-ID {}'.format(client_id)}
    req = Request('https://api.imgur.com/3/gallery/random/random/', headers=headers, method='GET')
    with urlopen(req) as resp:
        data = json.loads(resp.read().decode('utf-8'))
    return [item['link'] for item in data['data'] if 'type' in item and item['type'] in types]

def download_link(directory, link):
    download_path = directory / os.path.basename(link)
    with urlopen(link) as image, download_path.open('wb') as f:
        f.write(image.read())

def setup_download_dir():
    download_dir = Path('images')
    if not download_dir.exists():
        download_dir.mkdir()
    return download_dir
```

Task: fetch images from Imgur

```
from time import time

from download import setup_download_dir, get_links, download_link

CLIENT_ID = 'replace with your client ID'

def main():
    ts = time()
    download_dir = setup_download_dir()
    links = [l for l in get_links(CLIENT_ID)]
    for link in links:
        download_link(download_dir, link)
    print('Took {}s'.format(time() - ts))

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

Python multiprocessing module

- Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently.

Python multiprocessing module

- Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently.

- In multiprocessing, processes are spawned by creating a Process object and then calling its start() method.

```
#!/usr/bin/python

from multiprocessing import Process


def fun(name):
    print(f'hello {name}')

def main():

    p = Process(target=fun, args=('Peter',))
    p.start()


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

Python multiprocessing **join**

- In multiprocessing, processes are spawned by creating a Process object and then calling its start() method.

- The join method blocks the execution of the main process until the process whose join method is called terminates. Without the join method, the main process won't wait until the process gets terminated.

Python multiprocessing **join**

joining.py

```
#!/usr/bin/python

from multiprocessing import Process
import time

def fun():

    print('starting fun')
    time.sleep(2)
    print('finishing fun')

def main():

    p = Process(target=fun)
    p.start()
    p.join()


if __name__ == '__main__':

    print('starting main')
    main()
    print('finishing main')
```

```
$ ./joining.py
starting main
starting fun
finishing fun
finishing main
```

The *finishing main* message is printed after the child process has finished.

```
$ ./joining.py
starting main
finishing main
starting fun
finishing fun
```

When we comment out the join method, the main process finishes before the child process.

It is important to call the join methods after the start methods.

Python multiprocessing **Process Id**

- The os.getpid returns the current process Id, while the os.getppid returns the parent's process Id.

process_id.py

```
#!/usr/bin/python

from multiprocessing import Process
import os

def fun():

    print('-------------------------')

    print('calling fun')
    print('parent process id:', os.getppid())
    print('process id:', os.getpid())

def main():

    print('main fun')
    print('process id:', os.getpid())

    p1 = Process(target=fun)
    p1.start()
    p1.join()

    p2 = Process(target=fun)
    p2.start()
    p2.join()


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

```
$ ./parent_id.py
main fun
process id: 7605
-------------------------
calling fun
parent process id: 7605
process id: 7606
-------------------------
calling fun
parent process id: 7605
process id: 7607
```

The parent Id is the same, the process Ids are different for each child process.

Python multiprocessing **Pool**

- The management of the processes can be simplified with the Pool object. It controls a pool of processes to which jobs can be submitted. The pool's map method chops the given iterable into a number of chunks which it submits to the process pool as separate tasks.

- The map blocks the main execution until all computations finish.

- The Pool can take the number of processes as a parameter. It is a value with which we can experiment. If we do not provide any value, then the number returned by os.cpu_count() is used.

Python multiprocessing **Pool**

worker_pool.py

```
#!/usr/bin/python

import time
from timeit import default_timer as timer
from multiprocessing import Pool, cpu_count


def square(n):

    time.sleep(2)

    return n * n


def main():

    start = timer()

    print(f'starting computations on {cpu_count()} cores')

    values = (2, 4, 6, 8)

    with Pool() as pool:
        res = pool.map(square, values)
        print(res)

    end = timer()
    print(f'elapsed time: {end - start}')

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

```
$ ./worker_pool.py
starting computations on 4 cores
[4, 16, 36, 64]
elapsed time: 2.0256662130013865
```

On a computer with four cores it took slightly more than 2 seconds to finish four computations, each lasting two seconds.

```
$ ./worker_pool.py
starting computations on 4 cores
[4, 16, 36, 64, 100]
elapsed time: 4.029600699999719
```

When we add additional value to be computed, the time increased to over four seconds.

Python multiprocessing **Pool - Multiple functions**

multiple_functions.py

```
#!/usr/bin/python

from multiprocessing import Pool
import functools


def inc(x):
    return x + 1

def dec(x):
    return x - 1

def add(x, y):
    return x + y

def smap(f):
    return f()


def main():

    f_inc = functools.partial(inc, 4)
    f_dec = functools.partial(dec, 2)
    f_add = functools.partial(add, 3, 4)

    with Pool() as pool:
        res = pool.map(smap, [f_inc, f_dec, f_add])

        print(res)


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

We have three functions, which are run independently in a pool. We use the functools.partial to prepare the functions and their parameters before they are executed.

```
$ ./multiple_functions.py
[5, 1, 7]
```

This is the output.

Task: fetch images from Imgur

```
import json
import os
from pathlib import Path
from urllib.request import urlopen, Request

types = {'image/jpeg', 'image/png', 'image/gif'}

def get_links(client_id):
    headers = {'Authorization': 'Client-ID {}'.format(client_id)}
    req = Request('https://api.imgur.com/3/gallery/random/random/', headers=headers, method='GET')
    with urlopen(req) as resp:
        data = json.loads(resp.read().decode('utf-8'))
    return [item['link'] for item in data['data'] if 'type' in item and item['type'] in types]

def download_link(directory, link):
    download_path = directory / os.path.basename(link)
    with urlopen(link) as image, download_path.open('wb') as f:
        f.write(image.read())

def setup_download_dir():
    download_dir = Path('images')
    if not download_dir.exists():
        download_dir.mkdir()
    return download_dir
```

Python multiprocessing module

```
from time import time
from functools import partial
from multiprocessing.pool import Pool

from download import setup_download_dir, get_links, download_link

CLIENT_ID = 'replace with your client ID'

def main():
   ts = time()
   download_dir = setup_download_dir()
   links = [l for l in get_links(CLIENT_ID)]
   download = partial(download_link, download_dir)
   with Pool(8) as p:
       p.map(download, links)
   print('Took {}s'.format(time() - ts))

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