Motivation

- Python is readable
  - Easier to understand than Bash
- Python development is fast
- Python ecosystem is huge
  - Many data structures
  - Read/write data formats
  - Networking capabilities
  - Create visualizations

https://github.com/gjbex/Python-for-systems-programming/tree/master/source-code/command-line-arguments/
https://github.com/gjbex/Python-for-systems-programming/tree/master/source-code/config-parser

ICING ON APPLICATION: PYTHON'S ARGPARSE, CONFIGPARSER

Handling command line arguments

- Many tools start out as short script, evolve into applications used by many
- Model after Unix tools
  - Arguments
  - Flags
  - Options
- Python's `argparse` benefits
  - Easy to use
  - Self-documenting

Defining command line arguments

- Use `argparse` library module

```
from argparse import ArgumentParser
arg_parser = ArgumentParser(description='Gaussian random number generator')
```

- Add positional argument(s)

```
arg_parser.add_argument('nr', metavar='n', type=int, nargs='?', default=1,
                        help='number of random numbers to generate')
```

- Add flag(s)

```
arg_parser.add_argument('-idx', action='store_true', dest='index',
                        help='print index for random number')
```

- Add option(s)

```
arg_parser.add_argument('-mu', type=float, default=0.0,
                        help='mean of distribution')
```

`dest='mu'` is implicit

- Parse arguments

```
args = arg_parser.parse_args()
```

Using command line arguments

```
for i in range(args.nr):
    if args.index:
        prefix = f'{I + 1}\t'
    else:
        prefix = ''
    print(f'{args.mu}{args.sigma}')
```

```
$ ./generate_gaussians -h
usage: generate_gaussians.py [-h] [-mu MU] [-sigma SIGMA] [-idx] [n]
Gaussian random number generator
positional arguments:
  n              number of random numbers to generate
optional arguments:
  -h, --help     show this help message and exit
  -mu MU         mean of distribution
  -sigma SIGMA   stddev of distribution
  -idx           print index for random number
```

Autogenerated help message

```
$ ./generate_gaussians -idx 3.0
usage: generate_gaussians.py [-h] [-mu MU] [-sigma SIGMA] [-idx] [n]
generate_gaussians.py: error: argument n: invalid int value: '3.0'
```

ConfigParser configuration files

- Configuration files
  - save typing of options
  - Document runs of applications
- Easy to use from Python: configparser module
- Configuration file (e.g., '`test.conf`')

```
[physics]
# this section lists the physical quantities of interest
T = 273.15
N = 1
[meta-info]
# this section provides some meta-information
author = gjb
version = 1.2.17
```

section `physics`

section `meta-info`

`key = value`

comments

Note:
at least one section

Reading & using configurations

- Reading configuration file

```
from configparser import ConfigParser
cfg = ConfigParser()
cfg.read('test.conf')
```

- Using configuration values

```
temperature = cfg.getfloat('physics', 'T')
number_of_runs = cfg.getint('physics', 'N')
version_str = cfg.get('meta-info', 'version')
if cfg.has_option('physics', 'g'):
    acceleration = cfg.getfloat('physics', 'g')
else:
    acceleration = 9.81
```

Further reading: argparse

- Argparse tutorial
  https://docs.python.org/3/howto/argparse.html

https://github.com/gjbex/Python-for-systems-programming/tree/master/source-code/logging

LOGGING

Logging: motivation

- Useful to verify what an application does
  - in normal runs
  - in runs with problems
- Helps with debugging
  - alternative to print statements
- Various levels can be turned on or off
  - see only relevant output

Good practice

Initialize & configure logging

```
import logging
…
logging.basicConfig(level=level, filename=name, filemode=mode,
                    format=format_str)
…
```

- level: minimal level written to log
- filemode
  - 'w': overwrite if log exists
  - 'a': append if log exists
- format, e.g.,

  `'{asctime}:{levelname:{message}'`

Log levels

- `CRITICAL`: non-recoverable errors
- `ERROR`: error, but application can continue
- `WARNING`: potential problems
- `INFO`: feedback, verbose mode
- `DEBUG`: useful for developer

- User defined

Selecting log level

- CRITICAL
- ERROR
- WARNING
- INFO
- DEBUG

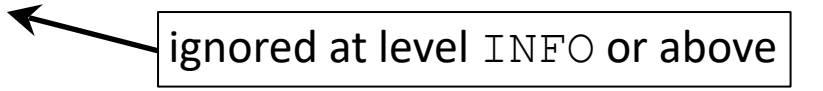
```
level = logging.ERROR
```

Log messages

- Log to `DEBUG` level

```
logging.debug(f'function xyz called with "{x}"')
```

- Log to `INFO` level




```
logging.info('application started')
```

- Log to `CRITICAL` level

```
logging.critical('input file not found')
```

Logging destinations

- File
- Rotating files
- syslog
- …

Further reading: logging

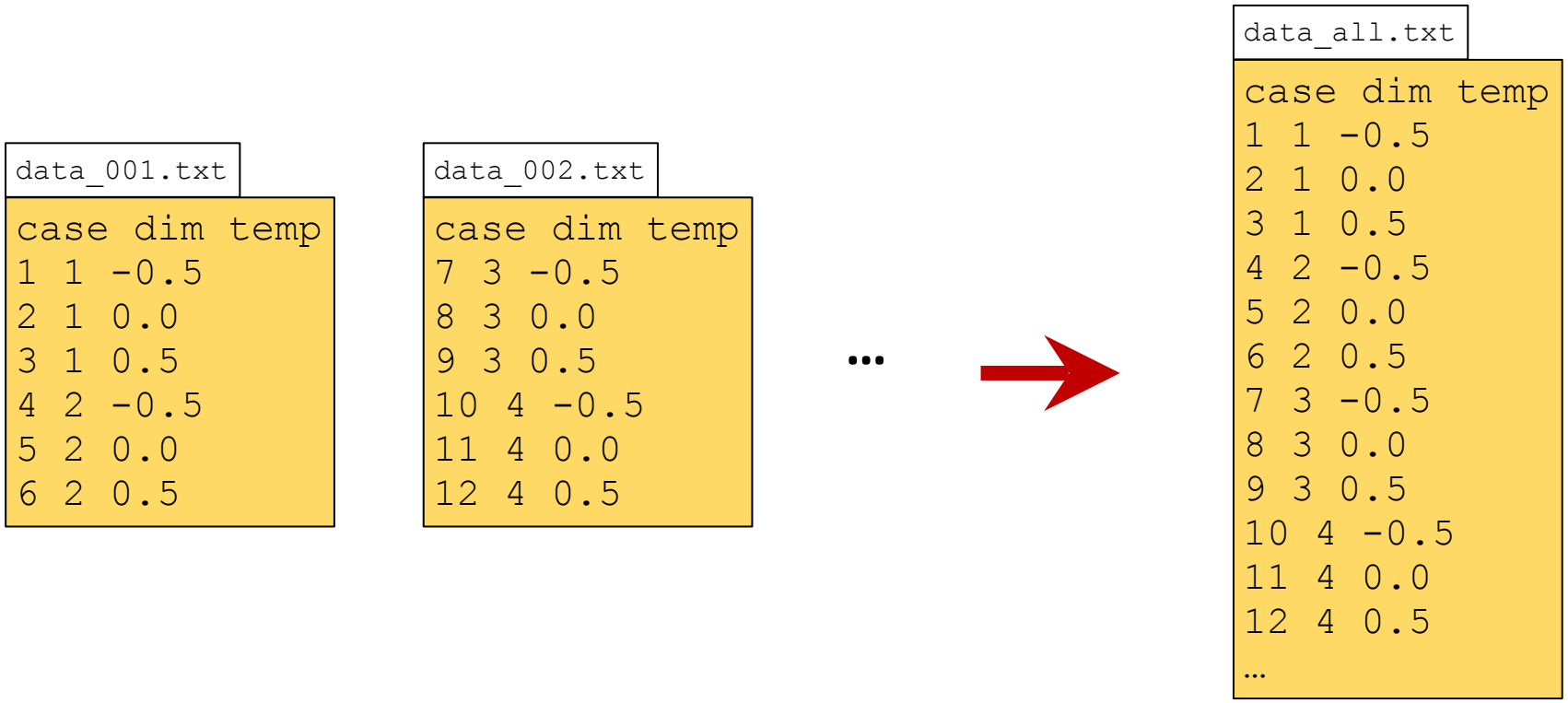
- Logging how-to
  https://docs.python.org/3/howto/logging.html
- Logging Cookbook
  https://docs.python.org/3/howto/logging-cookbook.html

https://github.com/gjbex/Python-for-systems-programming/tree/master/source-code/file-system

FILE SYSTEM OPERATIONS: HANDLING FILES AND DIRECTORIES

Working with files in directories

- Directory contains files `data_001.txt`, `data_002.txt`,…



Using glob

```
from argparse import ArgumentParser, FileType
from pathlib import Path
…
def main():
    arg_parser = ArgumentParser(description='…')
    arg_parser.add_argument('-o', dest='output_file',
                            type=FileType('w'), help='…')
    arg_parser.add_argument('-p', dest='pattern', help='…')
    options = arg_parser.parse_args()
    is_header_printed = False
    path = Path('.')
    for file_name in path.glob(options.pattern):
        with open(file_name, 'r') as input_file:
            header = input_file.readline()
            if not is_header_printed:
                options.output_file.write(header)
                is_header_printed = True
            for line in input_file:
                if line.strip():
                    options.output_file.write(line)
    return 0
```

Same as in Bash shell

```
$ python concat_data.py  -o data.txt  -p 'data_*.txt'
```

Path operations

- Many operations in `pathlib` package
  - Current working directory: `Path.cwd()`
  - Create path:

```
path = Path.cwd() / 'data' / 'output.txt'
    path == '/home/gjb/Tests/data/output.txt'
```

Will do the right thing for each OS

  - Dissecting paths:
    - `filename = path.name`
      `name == 'output.txt'`
    - `dirname = path.parent`
      `dirname == '/home/gjb/data'`
    - `parts = path.parts`
      `parts == ('/', 'home', 'gjb', 'data', 'output.txt')`
    - `ext = path.suffix`
      `ext == '.txt'`
    - `dirname = Path('/home/gjb/Tests').name`
      `dirname == 'Tests'`
    - `ext = Path('/home/gjb/Tests/').suffix`
      `ext == ''`

File system tests

- File tests:
  - `path.exists()`: `True` if `path` exists
  - `path.is_file()`: `True` if `path` is file
  - `path.is_dir()`: `True` if `path` is directory
  - `path.is_symlink()`: `True` if `path` is link
  - `pathlib.os.access(path, pathlib.os.R_OK)`: `True` if `path` can be read
    - `pathlib.os.R_OK`: read permission
    - `pathlib.os.W_OK`: write permission
    - `pathlib.os.X_OK`: execute permission

However: ask forgiveness, not permission!

Copying, moving, deleting

- Functions in `os` and `shutil` modules
  - copy file: `shutil.copy(source, dest)`
  - copy file, preserving ownership, timestamps: `shutil.copy2(source, dest)`
  - move file: `path.replace(dest)`
  - delete file: `path.unlink()`
  - remove empty directory: `path.rmdir()`
  - remove (non-empty) directory: `shutil.rmtree(directory)`
  - create directory: `path.mkdir()`

Temporary files

- Standard library `tempfile` package
  - Creating file with guaranteed unique name:
    `tempfile.NamedTemporaryFile(…)`

```
import tempfile
…
tmp_file = tempfile.NamedTemporaryFile(mode='w', dir='.',
                                       suffix='.txt', delete=False)
print(f"created temp file '{tmp_file.name}'"
with tmp_file.file as tmp:
    …
    tmp.write(…)
    …
```

File names such as `tmpD45x.txt`

Walking the tree

- Walking a directory tree: `os.walk(…)`, e.g., print name of Python files in (sub)directories

```
import os
…
for directory, _, file_names in os.walk(dir_name):
    directory = Path(directory)
    for file_name in file_names:
        file_name = Path(file_name)
        ext = file_name.suffix
        if ext == target_ext:
            print(directory / file_name)
…
```

- For each directory, tuple:
  - directory name
  - list of subdirectories
  - list of files in directory

For simple cases, use `path.rglob(…)`

https://github.com/gjbex/Python-for-systems-programming/tree/master/source-code/data-formats

DATA FORMATS

Libraries & data formats

- Standard library (Python 3.x)
  - Comma separated value files: `csv`
  - Configuration files: `ConfigParser`
  - Semi-structured data: `json`, `htmllib`, `sgmllib`, `xml`
- Non-standard libraries
  - Images: scikit-image
  - HDF5: pytables
  - pandas
  - Bioinformatics: Biopython

Use the "batteries" that are included!

Data formats: CSV

Let Sniffer figure out
CSV dialect (e.g., Excel)

```
 0  from csv import Sniffer, DictReader
 1  with open(file_name, 'rb') as csv_file:
 2      dialect = Sniffer().sniff(csv_file.read(1024))
 3      csv_file.seek(0)
 4      sum = 0.0
 5      csv_reader = DictReader(csv_file, fieldnames=None,
 6                              restkey='rest', restval=None,
 7                              dialect=dialect)
 8      for row in csv_reader:
 9          print(f'{row["name"]} --- {row["weight"]}')
10          sum += float(row['weight'])
11      print('sum = {0}'.format(sum))
```

DictReader uses first
row to deduce field names

Access fields by name,
thanks to DictReader

Drawback: you still need to know field types

Data formats: XML output

```
<?xml version="1.0" ?>
<blocks>
  <block name="block_01">
    <item>
      0.1
    </item>
    <item>
      1.1
    </item>
  </block>
  <block name="block_02">
    <item>
      0.2
    </item>
    <item>
      1.2
    </item>
  </block>
</blocks>
```

Data formats: creating XML

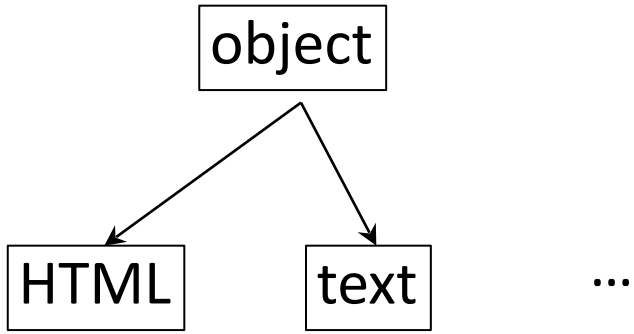
```
 1   from xml.dom.minidom import Document
 2   nr_blocks = 2
 3   nr_items = 2
 4   doc = Document()
 5   blocks = doc.createElement('blocks')
 6   doc.appendChild(blocks)
 7   for block_nr in range(1, nr_blocks + 1):
 8       block = doc.createElement('block')
 9       block_name = 'block_{block :02d}'
10       block.setAttribute('name', block_name)
11       blocks.appendChild(block)
12       for item_nr in range(0, nr_items):
13           item = doc.createElement('item')
14           text = f'{item_nr}.{block_nr}'
15           text_node = doc.createTextNode(text)
16           item.appendChild(text_node)
17           block.appendChild(item)
18   print(doc.toprettyxml(indent='  '))
```

https://github.com/gjbex/Python-for-systems-programming/tree/master/source-code/jinja

TEMPLATES

Separating information and representation

- Data as objects
  - represented as HTML/XML/…
- Code generation
  - wrappers
  - scripts



Data

- "Person" dict
  - ID
  - year of birth
  - number of friends

```
people = [
    {
        'id':          'YwaVW',
        'birthyear':   1954,
        'nr_friends':  42,
    },
    {
        'id':          'KfsaZ',
        'birthyear':   1952,
        'nr_friends':  22,
    },
]
```

HTML template

```
<table>
    <tr>
        <th>Person ID</th>
        <th>year of birth</th>
        <th>number of friends</th>
    </tr>
    {% for person in people %}
    <tr>
        <td> {{ person['id'] }} </td>
        <td> {{ person['birthyear'] }} </td>
        <td> {{ person['nr_friends'] }} </td>
    </tr>
    {% endfor %}
</table>
```

```
<table>
    <tr>
        <th>Person ID</th>
        <th>year of birth</th>
        <th>number of friends</th>
    </tr>
    <tr>
        <td> YwaVW </td>
        <td> 1954 </td>
        <td> 42 </td>
    </tr>
    <tr>
        <td> KfsaZ </td>
        <td> 1952 </td>
        <td> 22 </td>
    <tr>
    <tr>
        <td> HzyeL </td>
        <td> 1951 </td>
        <td> 32 </td>
    </tr>
    …
</table>
```

MarkDown template

```
| person ID | year of birth | number of friends |
|-----------|---------------|-------------------|
{% for person in people %}
| {{ '%-9s'|format(person['id']) }} | … | … |
{% endfor %}
```

```
| person ID | year of birth | number of friends |
|-----------|---------------|-------------------|
| YwaVW     |          1954 |                42 |
| KfsaZ     |          1952 |                22 |
…
```

Filling out templates

```
from jinja2 import Environment, PackageLoader
…
people = …
environment = Environment(loader=PackageLoader('population',
                                                'templates'),
                          trim_blocks=True, lstrip_blocks=True)
template = environment.get_template(f'report.{options.format}')
print(template.render(people=people))
```

- Create environment
- Load template
- Render template

https://github.com/gjbex/Python-for-systems-programming/tree/master/source-code/subprocess

USING SHELL COMMANDS: PYTHON SUBPROCESS

Counting words in a file

- Using shell utilities: `subprocess` module

```
$ wc text.txt
 4 12 52 text.txt
```

```
from subprocess import check_output
output = check_output(['wc', 'test.txt'])
output_str  = output.decode(encoding='utf-8')
lines, words, chars, _ = output_str.strip().split(' ')
```

Python 3 strings are unicode

- Convenient high-level API
  - `subprocess.call(…)` returns exit code of command as integer
  - `subprocess.check_output(…)` returns output of command as `bytes` (decode to get Python `str`)

Counting words in a string

- Low-level API: input & output

```
$ wc -
This is a single line.
      1       5      23 -
```

```
from subprocess import Popen, PIPE
text = bytes('This is a single line.\n', encoding='utf-8')
cmd = Popen(['wc', '-'], stdin=PIPE, stdout=PIPE)
cmd.stdin.write(text_str)
cmd.stdin.close()
output = cmd.stdout.readline().decode(encoding='utf-8')
lines, words, chars, _ = output.strip().split(' ')
```

Make sure `wc` *knows* it received all data!!!

`Popen(…, stdin=PIPE, stdout=PIPE)` creates file objects `stdin`/`stdout` for writing/reading, analogous to pipes in Unix

Remember, `stdin`/`stdout`/`stderr` use `bytes`!