Week 10 Laboratory Sample Solutions

Objectives

- Writing and use your own Python modules
- Exploring modules in Python

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Set up for the lab by creating a new directory called `lab10` and changing to this directory.

```
$ mkdir lab10
$ cd lab10
```

There are some provided files for this lab which you can fetch with this command:

```
$ 2041 fetch lab10
```

If you're not working at CSE, you can download the provided files as a zip file or a tar file.

EXERCISE:
DNA analysis in Python

Your task is to add code to the file `dna.py` to do DNA analysis.

Don't worry you don't need to know anything about DNA, RNA or base pairs.

You have been given the file `test_dna.py` that imports `dna.py` and uses its functions to analyse a file. Do not change `test_dna.py`. Only change `dna.py`

You have been given 6 test files `data1` .. `data6` containing base pairs (again don't worry you don't need to know what base pair is).

The format of the base pair files is simple:

```
$ sed -n 1,3p data1
G <-> C
T <-> A
A <-> T
```

But note one or both element of a base pair may be missing.

```
$ grep -E '^ <->|<-> $' data3|head
 <-> A
 <-> T
 <-> A
 <-> G
 <-> A
 <-> G
 <-> 
 <-> 
G <-> 
A <-> 
```

Here is how `test_dna.py` will work when you've completed the functions in `dna.py` (code>test_dna.py imports `dna.py`).

```
$ ./test_dna.py data1
the file data1 is DNA
there are 100 pairs in the file
first 10 pairs:
G <-> C
T <-> A
A <-> T
G <-> C
A <-> T
G <-> C
T <-> A
C <-> G
T <-> A
A <-> T
last 10 pairs:
C <-> G
T <-> A
G <-> C
T <-> A
G <-> C
G <-> C
T <-> A
C <-> G
A <-> T
T <-> A
the most common base is Guanine
```

The docstrings of the functions in `dna.py` give you more information about how to complete each function.

```
def read_dna(dna_file):
    """
    Read a DNA string from a file.
    the file contains data in the following format:
    A <-> T
    G <-> C
    G <-> C
    C <-> G
    G <-> C
    T <-> A
    Output a list of touples:
    [
        ('A', 'T'),
        ('G', 'C'),
        ('G', 'C'),
        ('C', 'G'),
        ('G', 'C'),
        ('T', 'A'),
    ]
    Where either (or both) elements in the string might be missing:
    <-> T
    G <->
    G <-> C
    <->
    <-> C
    T <-> A
    Output:
    [
        ('', 'T'),
        ('G', ''),
        ('G', 'C'),
        ('', ''),
        ('', 'C'),
        ('T', 'A'),
    ]
    """
    pass

def is_rna(dna):
    """
    Given DNA in the aforementioned format,
    return the string "DNA" if the data is DNA,
    return the string "RNA" if the data is RNA,
    return the string "Invalid" if the data is neither DNA nor RNA.
    DNA consists of the following bases:
    Adenine  ('A'),
    Thymine  ('T'),
    Guanine  ('G'),
    Cytosine ('C'),
    RNA consists of the following bases:
    Adenine  ('A'),
    Uracil   ('U'),
    Guanine  ('G'),
    Cytosine ('C'),
    The data is DNA if at least 90% of the bases are one of the DNA bases.
    The data is RNA if at least 90% of the bases are one of the RNA bases.
    The data is invalid if more than 10% of the bases are not one of the DNA or RNA bases.
    Empty bases should be ignored.
    """
    pass


def clean_dna(dna):
    """
    Given DNA in the aforementioned format,
    If the pair is incomplete, ('A', '') or ('', 'G'), ect
    Fill in the missing base with the match base.
    In DNA 'A' matches with 'T', 'G' matches with 'C'
    In RNA 'A' matches with 'U', 'G' matches with 'C'
    If a pair contains an invalid base the pair should be removed.
    Pairs of empty bases should be ignored.
    """
    pass

def mast_common_base(dna):
    """
    Given DNA in the aforementioned format,
    return the most common first base:
    eg. given:
    A <-> T
    G <-> C
    G <-> C
    C <-> G
    G <-> C
    T <-> A
    The most common first base is 'G'.
    Empty bases should be ignored.
    """
    pass

def base_to_name(base):
    """
    Given a base, return the name of the base.
    The base names are:
    Adenine  ('A'),
    Thymine  ('T'),
    Guanine  ('G'),
    Cytosine ('C'),
    Uracil   ('U'),
    return the string "Unknown" if the base isn't one of the above.
    """
    pass
```

Download dna.py, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs2041/22T2/activities/DNA/files.cp/dna.py dna.py
```

NOTE:

Your answer must be Python only. You can not use other languages such as Shell, Perl or C.

You must not use subprocess, or otherwise run external programs.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest DNA
```

When you are finished working on this exercise, you must submit your work by running `give` :

```
$ give cs2041 lab10_DNA dna.py
```

before **Monday 08 August 12:00** to obtain the marks for this lab exercise.

SOLUTION:

Sample solution for `dna.py`

```python
from collections import Counter, defaultdict

import sys


def read_dna(dna_file):
    """
    Read a DNA string from a file.
    the file contains data in the following format:
    A <-> T
    G <-> C
    G <-> C
    C <-> G
    G <-> C
    T <-> A
    The output is a list of touples
    [
        ('A', 'T'),
        ('G', 'C'),
        ('G', 'C'),
        ('C', 'G'),
        ('G', 'C'),
        ('T', 'A'),
    ]
    Where either element in the string might be missing.
    """
    data = []
    with open(dna_file) as f:
        for line in f:
            line = line.strip()
            base1, base2 = line.split("<->")
            data.append((base1.strip(), base2.strip()))

    return data

def is_rna(dna):
    """
    Given DNA in the aforementioned format,
    return "DNA" if the data is DNA,
    return "RNA" if the data is RNA,
    return "Invalid" if the data is neither DNA nor RNA.
    DNA consists of the following bases:
    Adenine  ('A'),
    Thymine  ('T'),
    Guanine  ('G'),
    Cytosine ('C'),
    RNA consists of the following bases:
    Adenine  ('A'),
    Uracil   ('U'),
    Guanine  ('G'),
    Cytosine ('C'),
    The data is DNA if at least 90% of the bases are one of the DNA bases.
    The data is RNA if at least 90% of the bases are one of the RNA bases.
    The data is invalid if more than 10% of the bases are not one of the DNA or RNA bases.
    """
    c = Counter()
    for pair in dna:
        c[pair[0]] += 1
        c[pair[1]] += 1

    if c['A'] + c['T'] + c['G'] + c['C'] >= (sum(c.values()) - c['']) * 0.9:
        return "DNA"

    if c['A'] + c['U'] + c['G'] + c['C'] >= (sum(c.values()) - c['']) * 0.9:
        return "RNA"

    return "Invalid"


def clean_dna(dna):
    """
    Given DNA in the aforementioned format,
    If the pair is incomplete, ('A', '') or ('', 'G'), ect
    Fill in the missing base with the match base.
    In DNA 'A' matches with 'T', 'G' matches with 'C'
    In RNA 'A' matches with 'U', 'G' matches with 'C'
    Any invalid pair is removed.
    """
    type = is_rna(dna)
    if type == "DNA":
        pairs = {'A': 'T', 'T': 'A', 'G': 'C', 'C': 'G', '': ''}
    elif type == "RNA":
        pairs = {'A': 'U', 'U': 'A', 'G': 'C', 'C': 'G', '': ''}

    new_dna = []
    for pair in dna:
        if pair[0] not in pairs.keys():
            continue
        if pair[1] not in pairs.keys():
            continue

        if pair[0] == '' and pair[1] == '':
            continue
        elif pair[0] == '':
            new_dna.append((pairs[pair[1]], pair[1]))
        elif pair[1] == '':
            new_dna.append((pair[0], pairs[pair[0]]))
        else:
            new_dna.append(pair)

    return new_dna

def mast_common_base(dna):
    c = Counter()
    for pair in dna:
        c[pair[0]] += 1

    return c.most_common(1)[0][0]

def base_to_name(base):
    name = defaultdict(lambda: "Unknown")
    name['A'] = "Adenine"
    name['T'] = "Thymine"
    name['G'] = "Guanine"
    name['C'] = "Cytosine"
    name['U'] = "Uracil"
    return name[base]
```

CHALLENGE EXERCISE:

Bashful Python

We have some Shell (Bash) scripts that do arithmetic calculations that we need to translate to Python.

Write a Python program `bashpy.py` which takes such a Bash script on stdin and outputs an equivalent Python program.

The scripts use the arithmetic syntax supported by Bash (and several other shells). Fortunately, the scripts only use a very limited set of shell features.

You can assume all the features you need to translate are present in the following 4 examples.

- `sum.sh` sums a series of integers:

```
$ cat sum.sh
#!/bin/bash

# sum the integers $start .. $finish

start=1
finish=100
sum=0

i=1
while ((i <= finish))
do
    sum=$((sum + i))
    i=$((i + 1))
done

echo $sum
$ ./sum.sh
5050
$ ./bashpy.py < sum.sh
#!/usr/bin/env python3
```

- `double.sh` prints powers of two:

```
$ cat double.sh
#!/bin/bash

# calculate powers of 2 by repeated addition

i=1
j=1
while ((i < 31))
do
    j=$((j + j))
    i=$((i + 1))
    echo $i $j
done
$ ./double.sh
2 2
3 4
4 8
5 16
6 32
7 64
8 128
```

- `pythagorean_triple.sh` searches for Pythagorean triples:

```
$ cat pythagorean_triple.sh
#!/bin/bash

max=42
a=1
while ((a < max))
do
    b=$a
    while ((b < max))
    do
        c=$b
        while ((c < max))
        do
            if ((a * a + b * b == c * c))
            then
                echo $a $b $c
            fi
            c=$((c + 1))
        done
        b=$((b + 1))
    done
```

- `collatz.sh` prints an interesting series:

```
$ cat collatz.sh
#!/bin/bash

# https://en.wikipedia.org/wiki/Collatz_conjecture
# https://xkcd.com/710/

n=65535
while ((n != 1))
do
    if ((n % 2 == 0))
    then
        n=$((n / 2))
    else
        n=$((3 * n + 1))
    fi
    echo $n
done
$ ./bashpy.py <collatz.sh
#!/usr/bin/env python3

# https://en.wikipedia.org/wiki/Collatz_conjecture
```

HINT:

Don't worry about the many possible features that can occur in shell scripts.
Focus on translating correctly the very limited set of Shell in the examples you are given.

For example, the Shell scripts you are given don't contain strings (quoted or unquoted), so don't worry about translating these.

You can translate each line of the shell script individually using simple string operations or `re.search` and `re.sub`.

NOTE:

You can assume the Shell scripts are always indented in the same manner as the example you've given

in other words the indenting will be compatible with Python.

You must translate the bash scripts to Python not just embed the shell of the script in a Python system call.

Your answer must be Python only. You can not use other languages such as Shell, Perl or C.

You must not use subprocess, or otherwise run external programs.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest bashpy
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs2041 lab10_bashpy bashpy.py
```

before **Monday 08 August 12:00** to obtain the marks for this lab exercise.

SOLUTION:

Sample solution for `bashpy.py`

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

import re
import sys


def main():
    for (line_number, line) in enumerate(sys.stdin):
        line = line.rstrip()
        if line_number == 0 and line.startswith("#!"):
            print("#!/usr/bin/env python3")
        else:
            translate_line(line)


def translate_line(shell):
    # remove and save leading white space
    indent = re.match(r"\s*", shell).group(0)
    shell = shell[len(indent) :]

    # remove and save any comment & trailing white space
    comment = re.search(r"\s*(#.*)?$", shell).group(0)
    shell = shell[: -len(comment) or None]

    python = translate_command(shell)

    if python is not None:
        print(indent + python + comment)


def translate_command(command):
    command = translate_expression(command)
    if command in ["do", "done", "fi", "then"]:
        return None
    if m := re.match(r"(if|while|else)", command):
        return command + ":"
    if m := re.match(r"echo\s+(.*)", command):
        arguments = m.group(1)
        arguments = re.sub(r"\s+", ", ", arguments)
        return f"print({arguments})"
    if m := re.match(r"(.*?)=(.*)", command):
        return f"{m.group(1)} = {m.group(2)}"
    return ""


def translate_expression(expression):
    expression = expression.replace('$', '')
    expression = expression.replace("))", '')
    expression = expression.replace("((", '')
    expression = expression.replace("/", "//")
    return expression


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

CHALLENGE EXERCISE:

When Regular Expressions Aren't Regular

Write a regular expression that validates a JSON file.

In other words, write a regex that matches a string iff (if and only if) that string is valid JSON.

Here is a test program assist you in doing this:

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

from sys import argv, stderr
import regex

regex.DEFAULT_VERSION = regex.V1

assert len(argv) == 3, f"Usage: {argv[0]} <json file> <regex file>"

json_file, regex_file = argv[1], argv[2]

try:
    with open(json_file) as json_data, open(regex_file) as regex_data:
        if regex.search(regex_data.read(), json_data.read(), timeout=5):
            # In the test suite, all files that start with "y_" should be valid.
            print(f"Valid   JSON file: {json_file}")
        else:
            # In the test suite, all files that start with "n_" should be invalid.
            print(f"Invalid JSON file: {json_file}")

except TimeoutError as e:
    # Allow a timeout error to signal that the jason file is not valid
    print(f"Invalid JSON file: {json_file}")
    # This is printed to stderr so that it is not captured by the test
    print(f"5 second time limit reached while reading {json_file}", file=stderr)
```

Download test_regex_json.py, or copy it to your CSE account using the following command:

```
$ cp -n /web/cs2041/22T2/activities/regex_json/test_regex_json.py test_regex_json.py
```

You have been given a directory `JSONTestSuite` containing a number of JSON files.

There are two types of files in this directory:
Files starting with `y_` are valid JSON files.
Files starting with `n_` are invalid JSON files.

Put your solution in `regex_json.txt` :

For example to test the regex `^.+$`

```
$ chmod 755 test_regex_json.py
$ unzip JSONTestSuite.zip
$ cat regex_json.txt
^.+$
$ ./test_regex_json.py JSONTestSuite/y_array_heterogeneous.json regex_json.txt
Valid   JSON file: JSONTestSuite/y_array_heterogeneous.json
$ ./test_regex_json.py JSONTestSuite/n_array_star_inside.json regex_json.txt
Valid   JSON file: JSONTestSuite/n_array_star_inside.json
# This should be Invalid so the regex is incorrect
```

If your solution is correct, all files in the `JSONTestSuite` starting with `y_` should be labelled valid, and all files starting with `y_` should be labelled invalid,

NOTE:

This exercise is not possible with true regular expression, i.e using `|*()` alone,
You will need to use additional regular expression syntax to achieve the same result.
Not even Python regular expression syntax enough.
You will need to use the PCRE regular expression library `regex` .
Autotest will use install ans use this module for you.
But you may need to install it manually for local testing.

The `regex` module extends the Python `re` module.
So the Python regular syntax documentation will still be useful.
You will additionally need to read the documentation for the `regex` module.

This is (another) *(in)?famous* problem to try and solve with regex.
There will be answers easily found online.
Don't google for other people solutions - see if you can come up with your own.

HINT:

You are not able to change the *flags=* argument of the `regex.search` function.
But there are ways to enable and disable flags within the regex itself.

-

Python supports the [`regex.VERBOSE` flag](#).
This flag allows you to add comments to your regular expression.
Plus whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash
This allows you to write much more readable regular expressions.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest regex_json
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs2041 lab10_regex_json regex_json.txt
```

before **Monday 08 August 12:00** to obtain the marks for this lab exercise.

SOLUTION:

Sample solution for `regex_json.txt`

```
(?six)

(?(DEFINE)
    (?P<keyword>   (?-i: true | false | null ) )
    (?P<number>    -? (?= [1-9]|0(?!\d) ) \d+ (\.\d+)? (e [+-]? \d+)? )
    (?P<string>    " ([^"\n\r\t\\]* | \\ ["\\bfnrt\/] | \\ u [0-9a-f]{4} )* " )
    (?P<array>     \[  (?:  (?&json)  (?: , (?&json)  )*  )?  [ \n\n\r\t]* \] )
    (?P<object>    \{  (?:  (?&pair)  (?: , (?&pair)  )*  )?  [ \n\n\r\t]* \} )
    (?P<pair>      [ \n\n\r\t]* (?&string) [ \n\n\r\t]* : (?&json)  )
    (?P<json>      [ \n\n\r\t]* (?: (?&number) | (?&keyword) | (?&string) | (?&array) | (?&object) )
[ \n\n\r\t]* )
)

\A (?&json) \Z
```

SOLUTION:

Alternative solution for `regex_json.txt`

```
(?six)
# enable 3 global flags
# `s` is DOTALL     - allow `.` to match newline
# `i` is IGNORECASE - match case insensitively
# `x` is VERBOSE    - allow whitespace and comments in regex
# this is part of python's `re` module, and doesn't require the `regex` module
# https://docs.python.org/3/library/re.html#index-16

# Note: that `(?x)` needs the be enabled on the first line of the regex as it only effects what
follows it
# And we might as well enable the others at the same time

# Note: IGNORECASE is only needed twice:
#   1. for scientific notation, (4.2e+10 / 4.2E+10)
#   2. for hexadecimal numbers, (0xff / 0XFF)
# We even have to turn off the IGNORECASE flag for keywords, which are case-sensitive
# So this might actully be more trouble than it's worth

# Note: we must use `[ \n\n\r\t]` instead of `\s` when we want to match whitespace
# this is because `\s` matches `\f` and `\v` but these are not allowed in JSON

# Note: the `(?P<word> ...)` syntax is used to create a named capture group
# This is part of the python `re` module, and doesn't require the `regex` module
# https://docs.python.org/3/library/re.html#index-17



# Note: the `(?&word)` syntax is a recursive reference to a named capture group
# This is added by the `regex` module.
# https://pypi.org/project/regex/#recursive-patterns-hg-issue-27
# This is different from the `(?P=word ...)` syntax, which is used as a backreference to a named
capture group
# https://docs.python.org/3/library/re.html#index-18
# The difference between a recursive reference and a backreference is:
#  1. recursive references - match the pattern using in the capture group
#  2. backreferences       - match the string that was previously matched by the capture group

# The `(?(DEFINE) ...)` syntax is added by the `regex` module.
# The `(?(DEFINE) ...)` syntax allows you to define sub-patterns for later use without them being
imidiatly executed.
# https://pypi.org/project/regex/#added-define-hg-issue-152
(?(DEFINE)
    # sub-pattern for matching a keyword
    # JSON has 3 keyword `true`, `false`, and `null`
    # keywords are case-insensitive, so we have to temporarily disable the IGNORECASE flag with `(?
i: ...)`
    (?P<keyword>   (?-i: true | false | null ) )

    # sub-pattern for matching a number
    # JSON numbers can be positive or negative (or zero), can have a decimal point, and can have a
scientific notation
    # JSON doesn't allow octal and hexadecimal formats, which makes our lives easier
    # JSON numbers can't have leading zeros, and can't have a leading decimal point
    # negative zero is allowed
    (?P<number>   -? (?= [1-9]|0(?!\d) ) \d+ (\.\d+)? (e [+-]? \d+)? )

    # sub-pattern for matching a string
    # JSON strings must be double-quoted, not single-quoted
    # JSON strings can contain any unicode character except double-quotes, backslashes, and controle
characters ASCII 0-31
    # JSON strings can contain escapted double-quotes or backslashes
    # JSON strings can contain the special characters `\b`, `\f`, `\n`, `\r`, `\t`
    # JSON strings can contain unicode escapes, which are of the form `\uXXXX` where `XXXX` is a 4-
digit hexadecimal number
    (?P<string>    " ([^"\n\r\t\\]* | \\ ["\\bfnrt\/] | \\ u [0-9a-f]{4} )* " )

    # sub-pattern for matching an array
    # JSON arrays must be enclosed in square brackets
    # JSON arrays can contain a comma seperated list of json documents
    (?P<array>     \[  (?:  (?&json)  (?: , (?&json)  )*  )?  [ \n\n\r\t]* \] )

    # sub-pattern for matching an object
    # JSON objects must be enclosed in curly braces
    # JSON objects can contain a comma seperated list of key value pairs
    (?P<object>    \{  (?:  (?&pair)  (?: , (?&pair)  )*  )?  [ \n\n\r\t]* \} )

    # sub-pattern for matching a key value pair
    # JSON key value pairs must be seperated by a colon
    # JSON key value pairs must contain a string for the key, and a json document for the value
    (?P<pair>      [ \n\n\r\t]* (?&string) [ \n\n\r\t]* : (?&json)  )

    # sub-pattern for matching a json document
    # a JSON document is any of the above sub-patterns
    # Note that some implementations only allow arrays or objects to be the top level of a JSON
document
    # But there is no actual limitation set in the JSON syntax so we allow all sub-patterns
    (?P<json>   [ \n\n\r\t]* (?: (?&number) | (?&keyword) | (?&string) | (?&array) | (?&object) ) [
\n\n\r\t]* )
)
# end of (?(DEFINE) ...)
# everything after this point will be executed imidiatly

# Do the actual match
# \A is the same as `^` and `\Z` is the same as `$`
# `(?&json)` is a reference to the previously defined sub-pattern
# so this line matches the an entire string as defined by the `json` sub-pattern
\A (?&json) \Z
```

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Week 11 Monday 12:00:00** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 2041 classrun -sturec
```