2019

SOFTENG 364: Labs 6 & 7 Orientation for Assignment 2

Craig Sutherland

Contents

Overview

Our primary goal is to become comfortable with the requirements of Assignment 2 and associated tools.

The number of activities in this worksheet is fairly large. Please be mindful of the time and don't get bogged down - speak with a member of the 364 team as soon as you have a query.

We won't attempt to complete Assignment 2 during lab time. Please work on Assignment 2 individually.

Conventions

- Lab tasks are marked with bullets, like this.

Notes have a blue marker on the edge

Preparation

We will use the same environment as Lab 5. If you are switching to a new PC, please follow the Preparation steps in the Lab 5 worksheet.

Assessment

To receive credit for completing the worksheet, please complete the Lab 6 & 7 Quiz on Canvas. You'll receive feedback immediately afterwards, and may choose to redo the quiz if you wish.

This worksheet and the associated quiz comprise Labs 6 & 7 and contribute 2% to the final mark. The due date (for the quiz) is the 5pm, Thursday 30th May. Nonetheless, please complete the worksheet and quiz at your earliest convenience so as to prepare for Assignment 2.

Part 1. Routing

This section relates to Part 1 of Assignment 2.

Predecessor maps

Task 1.1 involves updating Lab 1's `dijkstra_5_14()` to return the predecessor map (in addition to the optimal distances). This is reasonably straightforward and requires no new background.

Note that the resulting function should conform to the interface of NetworkX's `dijkstra_predecessor_and_distance()` i.e. having the same order- and types of output arguments.

Forwarding tables

Slide 19 of Lecture 1 displays the shortest-path forwarding table for Node u using the predecessor map computed on Slide 21 of Lecture 1. Task 1.2 asks us to automate the process.

Note: that we are not required to use an optimal algorithm and may visit each node more than once.

- Individually, determine the types of the input argument and output argument that would be consistent with dijkstra_predecessor_and_distance() and slide 5-17; then please move on with this worksheet.

Part 2. Error detection

This section relates to part 2 of Assignment 3.

The Internet Checksum

As part of Assignment 2, we'll implement the Internet Checksum algorithm. The steps in this section are intended to introduce the tools we'll require.

- Load IETF RFC 1071 (https://tools.ietf.org/html/rfc1071) on the Internet Checksum and, in not more than 5 minutes:
  - Read **Section 1** (for motivation) and **Section 4.1** (for a complete implementation in C).
  - Note that **Section 3** contains a numerical example - perhaps useful for debugging.
  - Skim read Property (P6) on [Page 16] and Section 4. on [Page 21] for a connection with CRCs.
  - Very quickly skim-read the headings of the rest of the document to conclude that there are various techniques a determined implementor can use to squeeze performance from the underlying hardware. We will not be concerned with performance.
- Have a quick look at http://www.netfor2.com/checksum.html, just to see what it contains. You may like to return to this document if you are unhappy with the one's complement sum.
- Read the example given in Wikipedia's description of the Internet Checksum (https://en.wikipedia.org/wiki/IPv4_header_checksum).

The Internet Checksum is not difficult to describe - and its implementation doesn't require many lines of code - but there are two issues that mightn't be clear at first:

1. How to manipulate bitstrings in Python.
2. How to extract bitstrings from arbitrary Python variables.

Python 3's integer type and bit manipulation

Note that we can express integer literals in binary and hexadecimal form:

```
>>> 0b1010  # binary
10
>>> 0xa     # hexadecimal
10
>>> assert 10 == 0b1010 == 0xa
```

- Introduce yourself to the built-in functions `bin()` and `hex()` by studying the following snippets.

```
>>> bin(10)
'0b1010'
>>> bin(10)[2:]
'1010'
>>> hex(10)
'0xa'
>>> hex(10)[2:]
'a'
```

- Why are these values returned as strings? Discuss with your neighbour. Although we won't need to do so here, it is also easy to go in the reverse direction:

```
>>> int('1010', base=2)
10
>>>
>>> int('a', base=16)
10
```

- Python 3's `int` type has unbounded size i.e. the size is constrained only by memory available on your computer. Try the following snippet to generate a 150-bit integer value; this is wider than the largest fixed-width quad-precision (128-bit) floating point values available on some platforms!

```
>>> n = 1 << 149
>>> n
```

```
713623846352979940529142984724747568191373312
>>> type(n)
<class 'int'>
>>> bin(n)
'0b10000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000'
>>> n.bit_length()
150
```

Java's `BigInt` class is similar.

- Briefly, discuss with your neighbor what the one's complement of an unbounded representation would look like. Can you interpret the behaviour of Python's **invert** operator ~?

```
>>> n = 0b10101111  # fits into one byte
>>> n
175
>>> ~n
-176
>>> n.bit_length()
8
>>> (~n).bit_length()
8
>>> bin(n)
'0b10101111'
>>> bin(~n)         # "has a sign?!"
'-0b10110000'
>>> bin(0b10110000)
'0b10110000'
```

Although ~ is, evidently, **not** the one's complement, it can be suitably modified: The trick is to apply a bitmask to recover the appropriate fixed-width representation.

```
>>> n = 0b10101111  # fits into one byte
>>> mask = 0xff     # one byte
>>> bin(n)
'0b10101111'
>>> bin(mask)
'0b11111111'
>>> bin(~n)         # not the bitwise complement
'-0b10110000'
>>> bin(~n & mask)  # is bitwise complement :)
'0b1010000'
```

We can use this formula in Task 2.1.

How might we obtain the bitstring that we know must underlie anything stored in our computer?

Python's bytes representation

Note: Although the methods of this section are relevant to network communication, they are not required for Assignment 2: Please don't spend more than a minute of two here.

We've seen that the Internet Checksum involves a treating an array of bytes as a sequence of 16-bit integers. We can obtain the sequence of bytes underlying Python's string types as:

```
>>> hello = 'Hello, Francis!'
```

```
>>> bonjour = 'Bonjour, Françoise!'
>>>
>>> hello.encode()
b'Hello, Francis!'
>>> hello.encode().decode()
'Hello, Francis!'
>>>
>>> bonjour.encode()
b'Bonjour, Fran\xc3\xa7oise!'
>>> bonjour.encode().decode()
'Bonjour, Françoise!'
```

- Where did the additional bytes in `bonjour` come from? Discuss with your neighbour. Note that Python 3 represents strings in Unicode (UTF-8) by default.
- To obtain the bytes underlying integer values, we use `to_bytes()`:

```
>>> x = 255
>>> x.to_bytes(2, byteorder='little')
b'\xff\x00'
>>> x.to_bytes(2, byteorder='big')
b'\x00\xff'
>>> x.to_bytes(4, byteorder='big')
b'\x00\x00\x00\xff'
>>> x.to_bytes(4, byteorder='little')
b'\xff\x00\x00\x00'
>>>
>>> try:
...     (1024).to_bytes(1, byteorder='little')
... except OverflowError:
...     print('int too big to convert')
...
int too big to convert
```

- Discuss this output with your neighbour. What is the significance of the `byteorder` parameter?

Byte ordering

Byte ordering might need to be considered when computing the Internet Checksum.

- Discuss the following output with your neighbour: What is the role of `socket.htons`?

```
>>> import socket
>>> x = 11
>>> bin(x)
'0b1011'
>>> bin(socket.htons(x))
'0b101100000000'
```

Versions of `hton()` are provided in the C network programming interface (`inet.h`) on all platforms.

CRC checks

This section relates to Task 2.3.

The CRC check explained on slide 6-15 relies on the XOR operator. In Python, as in C, C++, and Java, XOR is performed by operator-^:

```
>>> a = 0b1111
>>> b = 0b11110000
>>> bin(a ^ a)
'0b0'
>>> bin(a ^ b)
'0b11111111'
```

Unfortunately, Python's `int` type doesn't provide a means of indexing individual bits. Hence, we'll use the `bitarray` library: This is not part of the Pyopthon Standard Library, but might have been included with your Anaconda distribution; run the following line to check:

```
>>> import bitarray
```

If you see an error message, use `conda` to install `bitarray`:

```
> conda install bitarray
```

- Skim-read the description of key features listed on the project's GitHub page.

The library provides an efficient bitstring type, `bitarray.bitarray`, that works just as we might expect:

```
>>> from bitarray import bitarray
>>> a = bitarray('10101')
>>> b = bitarray([1, 0, 2, 0, 3])
>>> assert a == b
>>> a
bitarray('10101')
>>> b
bitarray('10101')
>>>
>>> c = bitarray(0b10101)
>>> c                                # Mmmm??
bitarray('011010001110011010111')
>>
>>> for bit in a:
...     print(bit)
...
True
False
True
False
True
>>> a[0]
True
>>> a[1]
False
>>>
```

- How did `bitarray` behave in the definitions of b and c? Discuss with your neighbour.

For Task 2.3, we'll write a function to implement a CRC check where the coefficients of the data- and generator polynomials are stored in bitarray.bitarray.

The following snippets may be helpful: Ensure that you can relate the output to the workings on slide 6-15.

```
def xor_at(a, b, offset=0):
```

```
    for k, bk in enumerate(b):
        index = offset + k
        a[index] = a[index] ^ bk

>>> from bitarray import bitarray
>>> D = bitarray('101110')  # data on slide 6-15
>>> G = bitarray('1001')    # generator on slide 6-15
>>>
>>> xor_at(D, G, 0)  # 1st step
>>> print(D)
bitarray('001010')
>>>
>>> xor_at(D, G, 2)  # 3rd step
>>> print(D)
bitarray('000011')
```

You might like to incorporate `xor_at` into Assignment 2, or you might prefer to start afresh: The only requirement is that you write to the interface `crc(data, generator)` and return the correct coefficients of the remainder in a `bitarray.bitarray`.

Part 3. ICMP

This section relates to Part 3 of Assignment 2.

Python's `namedtuple` container

We have already encountered (ordinary) tuples when functions return multiple outputs e.g. both a predecessor map and a distance map are returned from dijkstra_predecessor_and_distance.

Tuples are also provided by the standard libraries of C++, C# - but not Java or Go.

The following snippets to define Python tuples:

```
>>> t1 = 3, 'four'
>>> t2 = (3, 'four')
>>> t3 = tuple([3, 'four'])
>>> t1
(3, 'four')
>>> assert t1 == t2 == t3
```

- In this example, the parentheses were optional. Can you think of instances where they would be essential? Discuss with your neighbour.

Values are extracted by subscripting or destructuring:

```
>>> t = 1, 'two'
>>> a, b = t
>>> a
1
>>> b
'two'
>>> assert t[0] == a
>>> assert t[1] == b
```

These indices aren't very descriptive. For Task 3.1, we're asked to write a couple of utility functions to represent ICMP messages in `collections.namedtuple`s, which provide for self-documenting component extraction with no storage penalty. The following example demonstrates their use.

```
>>> import collections
>>> Complex = collections.namedtuple('Complex', ['real', 'imaginary'])
>>> z1 = Complex(1, 2)
>>> z2 = Complex(real=1, imaginary=2)
>>> z1
Complex(real=1, imaginary=2)
>>> assert z1 == z2
>>> assert z1[0] == z1.real
>>> assert z1[1] == z1.imaginary
>>> tuple(z1)
(1, 2)
```

- Is the order of the fields in an ICMP message important? Do you think `namedtuple` is a good fit? Discuss with your neighbour.

Python's `struct` module

We'll use Python's struct module to de/serialize ICMP message from/to Python types.

- The first two paragraphs of the documentation (https://docs.python.org/3/library/struct.html) explain the role of the struct library: Please read these carefully.

We'll need two of the module's functions: struct.pack() (for serialization) and struct.unpack (for deserialization). Both employ format specifier strings similar to those in `printf()` and `scanf()`, familiar from C and MATLAB.

```
import collections
Complex = collections.namedtuple('Complex', ['real', 'imaginary'])
z = Complex(1.2, 3.4)
type(z.real)
type(z.imaginary)
```

```
>>> import struct
>>> packet = struct.pack('ff', *z)
>>> packet
b'\x9a\x99\x99?\x9a\x99Y@'
>>>
>>> zz = Complex(*struct.unpack('ff', packet))
>>> zz
Complex(real=1.2000000476837158, imaginary=3.4000000953674316)
```

- Study the snippet above and discuss with your neighbour:
  - What types the format specifier ff represent?
  - What does struct.pack return?
  - What does struct.unpack return?
  - Why the use of operator-*?
  - Why do the components of zz differ from those of z in the 8th decimal place?

An implementation of ping

The final task of Assignment 2 is to implement Ping in Python using a raw socket and the functions we have written in Tasks 3.1 and 3.2. This is the most substantial part of Assignment 2.

We'll tackle the task as follows:

1. Parse command line arguments using the `argparse` module.

2. Define a new exception class: `ChecksumError`; you may reuse the Standard Library's `socket.timeout` or `TimeoutError` classes or define a replacement
3. Open a raw `socket.socket` in a with statement to ensure tidy termination
4. Specify a timeout on the socket, as specified at the command line
5. Create the ICMP header in binary format
6. Create the ICMP payload (the current time, one float) in binary format
7. Concatenate the ICMP header and payload using operator +
8. Send the ICMP echo request to the target host
9. Block on the echo reply
10. Handle time-out exceptions
11. Handle checksum errors
12. Estimate the echo's round-trip time
13. Compute and display ping statistics

Let's have a quick look at the items that we haven't yet encountered.

Command line arguments

Save the following lines in a new script called `hello.py`, say.

```
if __name__ == '__main__':

    import argparse
    parser = argparse.ArgumentParser(description='Extends a warm welcome.')
    parser.add_argument('-e', '--effusivity',
                        metavar='num',
                        type=int,
                        default=1,
                        help='Number of greetings.')
    parser.add_argument('names',
                        metavar='name',
                        type=str,
                        nargs='+',
                        help='Name of host or hostess.')
    args = parser.parse_args()

    for name in args.names:
        print('{}{}!'.format('Hello, ' * args.effusivity, name))
```

We can run our script from the Windows/Mac/Linux command line as follows:

```
> python hello.py
usage: hello.py [-h] [-e num] name [name ...]
hello.py: error: the following arguments are required: name

> python hello.py Alice
Hello, Alice!

> python hello.py Alice Ben
Hello, Alice!
Hello, Ben!

> python hello.py Alice Ben --effusivity 3
Hello, Hello, Hello, Alice!
```

```
Hello, Hello, Hello, Ben!

>python hello.py Alice Ben --e 3
Hello, Hello, Hello, Alice!
Hello, Hello, Hello, Ben!
```

- Study the program and discuss the following aspects with your neighbour:
  - Where did `parser` parse from?! There must be something happening behind the scenes?
  - Where did the usage string come from? i.e. `hello.py [-h] [-e num] name [name ...]`
  - What do the square brackets in the usage string mean? Where did they come from?
  - How are `-e, --effusivity`, and `args.effusivity` related?
  - Why use `argparser`? What does if offer?

Defining and using exceptions

An exception class must derive from `BaseException`, although it is recommended that we derive from `Exception` instead:

```
>>> class MyOwnError(Exception):
...     pass
...
>>> raise MyOwnError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyOwnError
>>>
>>> try:
...     raise MyOwnError()
... except Exception as error:
...     print('Caught an instance of {}'.format(type(error)))
...
Caught an instance of <class '__main__.MyOwnError'>
>>>
```

- Study the snippet and discuss with your neighbour:
  - What is the role of `pass`? What if we need to store information about the error condition?
  - How do the keywords `raise` and `except` compare with other languages you have used?
  - We caught an `Exception` as opposed to `MyOwnError`: Why might we have done this? When might we prefer to catch `MyOwnError`?

Open a raw `socket.socket`

We'll need the following snippet:

```
import socket
socket.socket(family=socket.AF_INET,
              type=socket.SOCK_RAW,     # <=="raw socket"
              proto=socket.getprotobyname('icmp'))
```

Most elements will be familiar from Assignment 1.

Using `with` statements

The following example and explanation are borrowed from the Python tutorial documentation:

```
>>>
>>> with open('workfile') as f:
...     read_data = f.read()
```

```
>>> f.closed
True
```

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using with is also much shorter than writing equivalent try-finally blocks

This advice extends to **sockets** and any other resource that the programmer is responsible for releasing.

To specify a timeout on the socket, you will need to use the method socket.settimeout().

Please be mindful of Python's built-in functions - many common operations are already implemented.

Best practice: Python coding conventions

Python's developers and maintainers have established a set of conventions for formatting Python source code, encoded in the PEP 8 - Style Guide for Python Code (https://www.python.org/dev/peps/pep-0008/).

- Skim-read this document very quickly, just to get a feel for what it covers. You might like to read it more carefully at your leisure.

PEP 8 is a good style to follow because:

1. Its conventions are not at all unreasonable;
2. They are well established and consistent with newer parts of the Standard Library;
3. The utility `pycodestyle` can check your code against some of the conventions in PEP 8.

Visit the pycodestyle documentation to see several examples of usage and output (http://pycodestyle.readthedocs.io/en/latest/index.html).

Here is small example of a file that doesn't conform to PEP 8 conventions:

```
import pprint, os
def f(x):
  return x # Comment 1

def g(x):
  return f(f(x)) # Comment 2
```

Here is a list of problems detected by `pycodestyle`:

```
> >pycodestyle my_module.py
my_module.py:1:14: E401 multiple imports on one line
my_module.py:2:1: E302 expected 2 blank lines, found 0
my_module.py:3:3: E111 indentation is not a multiple of four
my_module.py:3:11: E261 at least two spaces before inline comment
my_module.py:5:1: E302 expected 2 blank lines, found 1
my_module.py:5:10: W291 trailing whitespace
my_module.py:6:3: E111 indentation is not a multiple of four
my_module.py:6:17: E261 at least two spaces before inline comment
my_module.py:6:29: W292 no newline at end of file
```

- Run pycodestyle on the script(s) you have written for today's lab and make any adjustments necessary to conform to the conventions of PEP 8.

Please use pycodestyle to check the code you write for Assignment 2. Conformance with PEP 8 will contribute to the Best Practice component of the marking rubric.

End of lab worksheet: Please don't forget to complete the lab quiz.