SIRE504
Introduction to Python – part 2

11.10.2018

Harald Grove

harald.gro@mahidol.ac.th

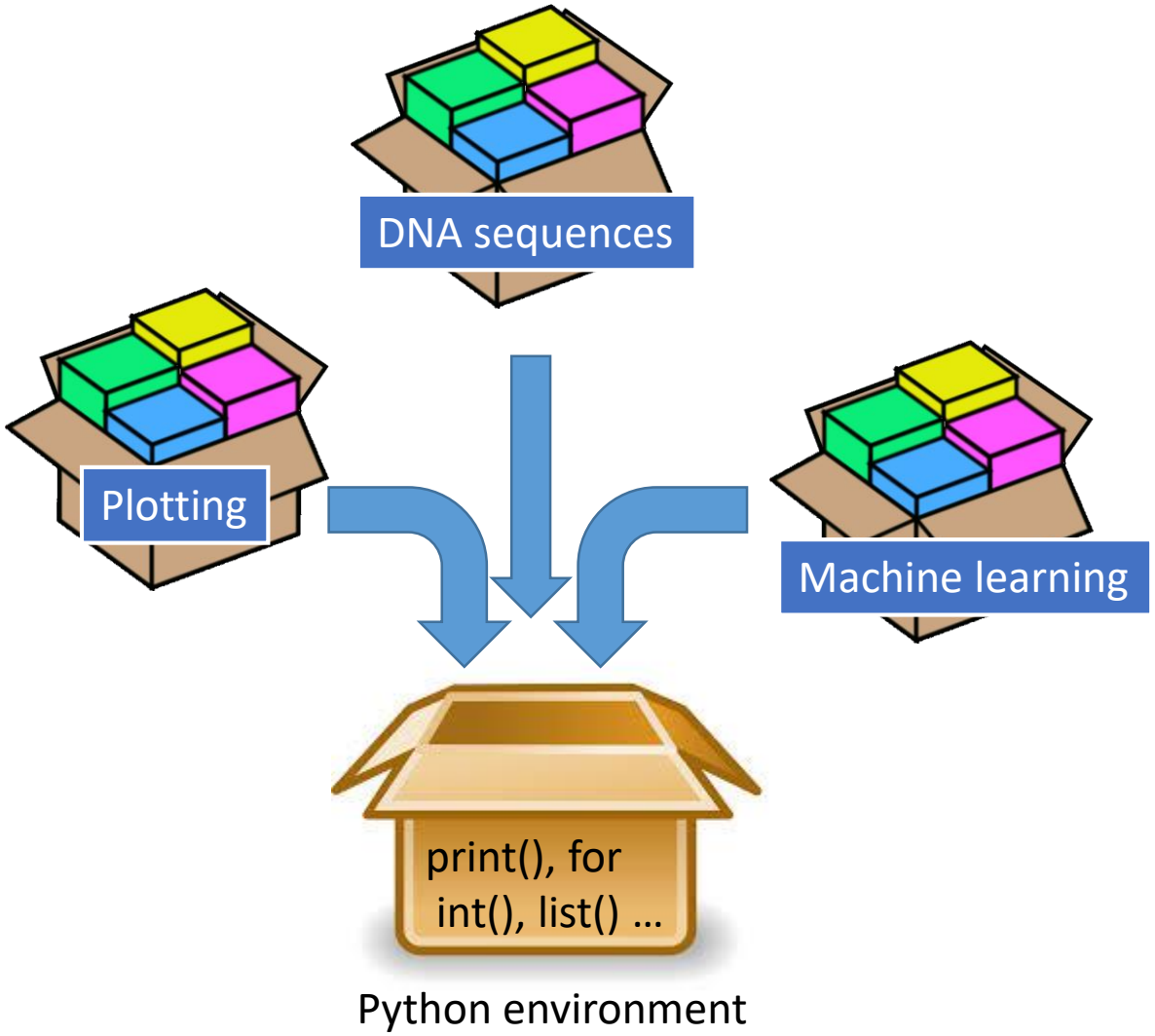
Topics

- Importing modules
- Command line interpreting
- Text files, reading and writing
- Dictionaries
- Functions

Modules

Importing Modules

- Core Python contains a limited number of methods and objects
  - Saves on memory
- Extra functionality can be added with modules



Importing syntax

- Import the whole module:
  - import <module name>
    - import pandas
    - import Bio
- Only import a specific object
  - from <module name> import <method>
    - from Bio import SeqIO
  - NB! This does not import the module itself
- Option to save some typing
  - import pandas as pd

Working with modules

- Access methods in modules:
  - <module>.<method>
- Example: using the “pi” variable in the “math” module

```
>>> import math
>>> math.pi
3.141592653589793
```

- Or

```
>>> from math import pi
>>> pi
3.141592653589793
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

Example project

Project: read and manipulate fasta files

- Specifications:
  - Read sequences from a fasta file
  - Write reports
  - User can specify file name when calling program
  - User can specify action to take after reading sequences
- We need:
  - Read and write (text) files
  - Interpret (parse) the command line
  - Select what to do based on input
  - String handling

The input file

Header line, starts with '>'

- dengue1.fna:

```
>U88536.1 Dengue virus type 1 clone 45AZ5, complete genome
AGTTGTTAGTCTACGTGGACCGACAAGAACAGTTTCGAATCGGAAGCTTGCTTAACGTAGTTCTAACAGTTTTTTATTAG
AGAGCAGATCTCTGATGAACAACCAACGGAAAAAGACGGGTCGACCGTCTTTCAATATGCTGAAACGCGCGAGAAACCGC
GTGTCAACTGTTTCACAGTTGGCGAAGAGATTCTCAAAAGGATTGCTTTCAGGCCAAGGACCCATGAAATTGGTGATGGC
TTTTATAGCATTCCTAAGATTTCTAGCCATACCTCCAACAGCAGGAATTTTGGCTAGATGGGGCTCATTCAAGAAGAATG
GAGCGATCAAAGTGTTACGGGGTTTCAAGAAAGAAATCTCAAACATGTTGAACATAATGAACAGGAGGAAAAGATCTGTG
ACCATGCTCCTCATGCTGCTGCCCACAGCCCTGGCGTTCCATCTGACCACCCGAGGGGGAGAGCCGCACATGATAGTTAG
CAAGCAGGAAAGAGGAAAATCACTTTTGTTTAAGACCTCTGCAGGTGTCAACATGTGCACCCTTATTGCAATGGATTTGG
GAGAGTTATGTGAGGACACAATGACCTACAAATGCCCCCGGATCACTGAGACGGAACCAGATGACGTTGACTGTTGGTGC
AATGCCACGGAGACATGGGTGACCTATGGAACATGTTCTCAAACTGGTGAACACCGACGAGACAAACGTTCCGTCGCACT
GGCACCACACGTAGGGCTTGGTCTAGAAACAAGAACCGAAACGTGGATGTCCTCTGAAGGCGCTTGGAAACAAATACAAA
AAGTGGAGACCTGGGCTCTGAGACACCCAGGATTCACGGTGATAGCCCTTTTTCTAGCACATGCCATAGGAACATCCATC
```

…

DNA sequence, arbitrary number of lines

Text file handling

Read and write

Read files

- First create a file object from the file
  - `>>> inputfile = open("dengue1.fna","r")`
  - The “r” makes the file object read-only
  - This object is iterable, with each line being the individual elements
  - We can use a for loop to access
  - When finished, use the method “close” to disconnect and clean up.

Reading files

- Files can be read line by line:

```
inputfile = open(“dengue1.fna”,”r”)
for line in inputfile:
    print(line)
inputfile.close()
```

- Take note of the line shifts in the output.

Write files

- First create a file object from the file
  - >>> outputfile = open('dengue1_mod.fna', 'w')
  - The 'w' creates an empty file, and the file object has write access
  - NB! Any existing files with the same name will be lost!
- Send content to the file with the method "write":
  - outputfile.write(<string>)
  - "write" is less flexible than "print"
  - <string> must be a single text variable, pre-formatted exactly the way it should appear.

Our program

```
inputfile = open(“dengue1.fna”,”r”)
outputfile = open(“dengue1_mod.fna”,”w”)
for line in inputfile:
    fout.write(line)
inputfile.close()
outputfile.close()
```

Next: user input

- Our script doesn’t do very much, just copies a text file.
- We want more functionality
- User should provide the names of the input and output
- Later on, user should also select the action to take

Command line parameters

The command line

- Passing parameters on the command line.
  - `$ bwa mem reference.fasta sample.1.fq sample.2.fq`
- Python has a module called “sys” to do the same
- Adding the module to your python environment:
  - import sys
- The full command line is stored in a list:
  - argv
- Use:

```
import sys
print(sys.argv)
```

The content of sys.argv

greeting.py:

```
import sys
print(sys.argv)
```

- Command:
  - `$ python greeting.py file1 file2 10 20`
- Output:
  - `['greeting.py', 'file1', 'file2', '10', '20']`
  - A list with (only) string elements
  - The name of the program is always the first element: sys.argv[0]

Get file names from user

- We want to get the file names from the command line
  - First parameter: input file name
    - inputfilename = sys.argv[1]
  - Second parameter: output file name
    - outputfilename = sys.argv[2]

Our program

```
import sys
inputfilename = sys.argv[1]
outputfilename = sys.argv[2]
inputfile = open(inputfilename,”r”)
outputfile = open(outputfilename,”w”)
for line in inputfile:
    fout.write(line)
inputfile.close()
outputfile.close()
```

Add functionality

- Let user decide what to do
  - Another parameter, but let the action be the first in the list
  - First parameter: action
  - Second parameter: input file name
  - Third parameter: output file name
- Actions:
  - count A, C, G, T and N
  - in case of non-ACGTN, replace with N and write a new file
  - search for a specific pattern, and report positions
  - and more … ?

Improve the data management

- We may need to have repeated access to our sequences
  - Reading the file every time could be time consuming
- We can store the name and the sequence as two strings
- We can connect them by using a new data type: dict()

Dictionaries

- Dictionary (dict) stores data in a key – value format
  - Unlike lists which stores values by position
- The value is accessed by providing the key
- dicts are iterable and the keys can be looped over
  - Don’t assume that the keys will come out in a specific order by itself
  - From Python v3.6, the input order is preserved

Dictionaries

- Create a new dictionary:
  - `>>> database = {<key1>:<value>, …}`
  - `>>> database = {"Thailand":"Bangkok", "France":"Paris"}`
- Add elements to an existing dictionary:
  - `>>> database["Canada"] = "Ottawa"`
- Access elements of a dictionary:
  - `>>> database["Thailand"]`
- Loop over elements:

```
for key in database:
    print(key)
for key, value in database.items():
    print(key, value)
```

Special assignment

Aside: Advanced assignments

- Multiple assignments

```
>>> line = [0,1,2]
>>> a,b,c = line
```

- Chained assignments

```
>>> a = b = 1
```

- Advanced multiple assignments (Python 3)

```
>>> a, b, *c = 0, 1, 2, 3, 4, 5, 6
```

  - a becomes '0'
  - b becomes '1'
  - c becomes a list of anything that is left '[2, 3, 4, 5, 6]'

Back to our program

- We want to store our sequences in a dict
  - The sequence name is the “key”
  - The sequence is the “value”
- We need to identify what we read from the file
  - Fasta format has two types of lines:
    - header line: starts with ‘>’
    - sequence line: any other line
- Actions:
  - Header line: create new entry in the database, with empty sequence
  - Sequence line: add to the sequence in the database

Our program

```
import sys

action = sys.argv[1]
inputfilename = sys.argv[2]
outputfilename = sys.argv[3]

inputfile = open(inputfilename,'r')
database = {}
for line in inputfile:
    if line.startswith('>'):
        seqname = line[1:].strip()
        database[seqname] = ''
        continue
    database[seqname] = database[seqname] + line.strip()
inputfile.close()

outputfile = open(outputfilename,'w')
for key, value in database.items():
    output = '>{}\n{}\n'.format(key,value)
    outputfile.write(output)
outputfile.close()
```

Parsing command line

Reading in fasta file and storing the data

Writing out the stored data

Action1: count bases

The first action: counting

- We first need to detect what the user wants:
  - Define keywords the user can use, i.e. “count”
- Only run code that belong to this action
  - We need to split the code into “blocks”
- Counting can be done with the string method “count()”
  - sequence.count(‘A’)
  - we decide we want the report to look like this:

    sequence name
    A: 100
    C: 100
    G: 100
    T: 100

Our program, new code

Beginning of file is the same: parsing command line and reading data.

```
…
outputfile = open(outputfilename,'w')
if action == 'count':
    bases = ['A', 'C', 'G', 'T', 'N']
    for name, sequence in database.items():
        outputfile.write('{}\n'.format(name))
        for base in bases:
            count = sequence.count(base)
            outputfile.write('{}: {}\n'.format(base, count))
else:
    for key, value in database.items():
        output = '>{}\n{}\n'.format(key,value)
        outputfile.write(output)
outputfile.close()
```

If user has chosen ‘count’, count each base and print the report.

If user has chosen anything else, just write the sequences to the output file.

Action2: clean fasta sequence

Replace bases

- String has a method to replace one substring with another
  - >>> sequence.replace(‘X’, ‘N’)
- But how do we discover which symbols needs to be replaced?
  - We could check each base and record each non-ACGTN base we discover.
  - In this case we would just replace the base at the same time.
- A procedural solution:
  - Loop through your sequence, one base at the time, checking each base if it’s a valid letter and replace it if it’s not.

Our program, new code

```
…
elif action == 'clean':
    bases = ['A', 'C', 'G', 'T', 'N']
    for name, sequence in database.items():
        newsequence = ''
        for base in sequence:
            if base not in bases:
                base = 'N'
            newsequence += base
        database[name] = newsequence
…
```

Add this section after the “count” block, before “else”.

Action3: search for patterns

Search for a pattern

- Looking for k-mers is an important function
- Let the user specify a (short) sequence as a command line parameter
- Report if the sequence is present in the fasta file and if so, where.

Searching with Python

- It’s easy to check if a string is present in another string

```
if kmer in sequence:
   print(“yes”)
else:
   print(“no”)
```

- How to get the coordinates?
  - str.find(pattern, start=0)

```
>>> sequence.find(‘ATG’)
```

  - Returns the coordinate of S where the start of pattern matches
  - If not found, returns -1
  - NB! Remember that ‘-1’ is a valid index!

String method “find”

- sequence = ‘acgacgacg’
- Default settings:

```
>>>sequence.find(‘ac’)
0
```

- Specify start position for search (first match starting from “start”):

```
>>>sequence.find(‘ac’,1)
3
```

- Search is case sensitive and returns “-1” when no match was found.

```
>>>sequence.find(‘AC’)
-1
```

New requirements

- Additional command line parameter: search pattern
- Search each sequence for the pattern
  - The pattern might occur several times

Our program, new code

```
…
elif action == 'search':
    pattern = sys.argv[4]
    for name, sequence in database.items():
        outputfile.write('{} :'.format(name))
        index = -1
        while True:
            index = sequence.find(pattern, index + 1)
            if index != -1:
                outputfile.write(' {}'.format(index))
            else:
                break
        outputfile.write('\n')
…
```

Search pattern is specified on the command line as another parameter.

This loop will never quit by itself.

Exits the current loop (while …), but not the outer loop (for …)

Clean up the code

- Make our own functions
- Method: a function that belongs to an object.
- Code that belongs together and does a specific (small) task can be “hidden” away in function.
- Main part of the code becomes more readable and easier to understand.
- Code for an action that is used many places can be moved to one place.
  - Makes updates easier.

Functions

- Syntax:

```
>>>def function_name(param1, param2, …):
>>>     <indented code block>
```

- Call the function:

```
>>>function_name(param1, param2, …)
```

- Recommended: pass all information the function requires through the provided interface.

Our program, new code

Code is packed away in functions

```
…
if action == 'count':
    for name, sequence in database.items():
        countbases(name, sequence, outputfile)
elif action == 'clean':
    for name, sequence in database.items():
        database[name] = cleansequence(name, sequence)
elif action == 'search':
    pattern = sys.argv[4]
    for name, sequence in database.items():
        searchpattern(name, sequence, pattern, outputfile)
…
```

```
def countbases(name, sequence, outputfile):
    bases = ['A', 'C', 'G', 'T', 'N']
    outputfile.write('{}\n'.format(name))
    for base in bases:
        count = sequence.count(base)
        outputfile.write('{}: {}\n'.format(base, count))


def cleansequence(name, sequence):
    bases = ['A', 'C', 'G', 'T', 'N']
    newsequence = ''
    for base in sequence:
        if base not in bases:
            base = 'N'
            newsequence += base
    return newsequence

def searchpattern(name,sequence, pattern, outputfile):
    outputfile.write('{} :'.format(name))
    index = -1
    while True:
        index = sequence.find(pattern, index + 1)
        if index != -1:
            outputfile.write(' {}'.format(index))
        else:
            break
    outputfile.write('\n')
```

Modules

Advanced command line parsing

- Module argparse:
- Specify which options you want.
- The module will take care of splitting the command line and formatting the input according to your design.
- It also provides a readable help text for the user (“-h”).

Argparse demo

```
$ python argparse_demo.py -h
usage: argparse_demo.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

```
import argparse


parser = argparse.ArgumentParser(description="calculate X to the
power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y


if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y,
answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

Sequence handling

- BioPython (https://biopython.org/)
- From the docs, a usage example on Orchids:
  - Parse sequences from any fasta file
  - Search PubMed and extract sequence data from GenBank
  - Extract protein data from Swiss-Prot
  - Work with ClustalW for multiple sequence alignment

```
>>> from Bio import SeqIO
>>> for record in SeqIO(fileobject, 'fasta'):
```

Defensive programming

- Split large tasks into smaller sub-tasks.
- Functions can be a good way to do this.
- Test each sub-task properly before moving on to the next.
- Mistakes are easier to find and correct when the amount of code is managable.

Defensive programming

- Types of errors:
  - Syntax errors
    - Stops the program from running.
    - Easiest to detect.
  - Run-time errors
    - Only occurs when the program is running, may not appear for all types of inputs.
    - This error is commonly "found" by the user, since it is often connected to non-standard input.
  - Logical errors
    - The program seems to work, even producing output. However, the output is not correct.
    - Most difficult to detect.

Python syntax

A more detailed look at the syntax of Python functions and methods

Modules

- Import the whole module
  - import <modulename>
  - Refer to methods as: modulename.methodname()
- Import specific parts of a module
  - from <modulename> import <methodname>
  - Refer to method as <methodname>
- Shorten name of module on import
  - import <modulename> as <short_modulename>

Tuples

- Same datatype as lists, but not mutable.
  - Syntax: (v0, v1, …., vn)
  - Tuples use parantheses "()", lists use square brackets "[]"
- Mostly used when you want a list that should not be allowed to change.

Advanced assignments

- Multiple assignments
  - a,b = 0,1
- Chained assignments
  - a = b = 1
- Advanced multiple assignments (Python 3)
  - a, b, *c = 0, 1, 2, 3, 4, 5, 6
  - a becomes '0'
  - b becomes '1'
  - c becomes a list of anything that is left '[2, 3, 4, 5, 6]'

Functions and recursion

- Sometimes we have code we want to use several places
  - In-built functions: round(), split(), etc.
- Syntax for creating our own functions
  - def <name>(<input1>, …, <input2>):
    - Statements
    - Return statement, if needed
- Local and global variables
  - Variables on the outside are visible inside of functions
  - The opposite is not true.
- Recursion
  - Functions can be called from inside of other functions
  - Functions can also call themselves.

Dictionaries

- Contain collection of object.
  - Objects can be any python object: number, string, list, dictionary, file handle…
- Data is stored in a "key:value" format.
  - Keys cannot be mutable (e.g. lists)
  - Value is the object we want to store
  - Examples: Country:Capital, Term:count
- Access data:
  - value = D[key]
  - value = D.get(key [, default_value])
- Remove data:
  - del D[key]

Dictionaries, part 2

- Check for presence/absence of a key in a dictionary:
  - key in D
  - key not in D
- Iterating over a dictionary:
  - for key in D:
  - for key in D.keys():
  - for value in D.values():
  - for key, value in D.items():

File input/output

- Reading files:
  - Establish connection to the file: fobj = open(filename, action)
  - Action is one of write (w), read (r), append (a).
  - The 'open' function returns a file object.
- The file object is iterable
  - Loop over the lines using
    - for line in file-object:
- Other options to read file
  - Read complete file as one long string: s = fobj.read()
  - Read complete file as a list containing each line is a string: L = fobj.readlines()
  - Read next line as a string: s = next(fobj)

File input/output, part 2

- Writing to file
  - Write one string at the time: fobj.write(s)
- Closing file
  - When the program is finished reading/writing the file, the connection should be closed:
    - fobj.close()
  - Closing the file is important when writing to the file, since data is not sent directly to the file, but is kept in a buffer.
- Easy approach to managing open/closen files:
  - with open(filename, 'r') as fin:
    - Statements
  - Benefit to this method is that python will take care of closing the file for you, even if the program ends unexpectedly.

File input/output, part 3

- Easy approach to managing open/closen files:

```
with open(filename, 'r') as fin:
    <statements>
```

- Benefit of this method is that python will take care of closing the file for you, even if the program ends unexpectedly.

Command line options

- The sys module (import sys) contains a list with the contents of the command line used to run the program.
  - sys.argv
- Command line:
  - $ python myscript.py infile.txt –d 10 –a 5
- The list sys.argv will now contain:
  - ['myscript.py', 'infile.txt', '-d', '10', '-a', '5']
  - Elements are separated by whitespace (space or tab)
  - All objects in the list are strings.