Fenna Feenstra
Hanze *University of Applied Sciences*
Groningen

Bioinformatics data processing and –analysis using the scripting language Python

Introduction to programming with Python

Functions and flow

Control flow recap

there are specific control flows in python

- the if statement
- the for loop
- the while loop
- the break statement
- the continue statement

if statement

- The if statement is used to check a condition: if the condition is true, we run a block of statements (called the if-block), elif we process the elif block (called the elif-block), else we process the else block of statements (called the else-block).
- The else clause and elif clause are optional. The if, elif and else blocks are much more efficient concequenced if statements since when a statement is true it will skip the others, which saves time.

if statement

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
elif guess < number:
    # Another block
    print('No, it is a little higher than that')
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guessed > number to reach here

print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

```
Enter an integer : 24
No, it is a little lower than that
Done
```

while statement

- The while statement allows you to repeatedly execute a block of statements as long as a condition is true. A while statement is an example of what is called a looping statement. A while statement can have an optional else clause.

while statement

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

```
Enter an integer : 24
No, it is a little lower than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

for loop

- The for..in statement is another looping statement which iterates over a sequence of objects i.e. go through each item in a sequence.

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

```
1
2
3
4
The for loop is over
```

break statement

- The break statement is used to break out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become False or the sequence of items has not been completely iterated over. An important note is that if you break out of a for or while loop, any corresponding loop else block is not executed.

break statement

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    print('Length of the string is', len(s))
print('Done')
```

```
Enter something : 33
Length of the string is 2
Enter something : 3333
Length of the string is 4
Enter something : 33
Length of the string is 2
Enter something : quit
Done
```

continue statement

- The continue statement is used to tell Python to skip the rest of the statements in the current loop block and to continue to the next iteration of the loop.

continue statement

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
        continue
    print('Input is of sufficient length')
    # Do other kinds of processing here...
```

```
Enter something : 33
Too small
Enter something : 3333
Input is of sufficient length
Enter something : 3333
Input is of sufficient length
Enter something : 333
Input is of sufficient length
Enter something : quit
```

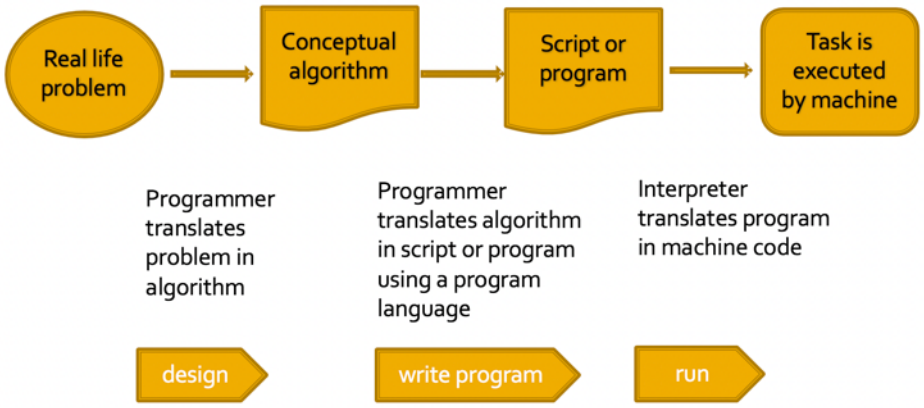
How to start your own script

How to start your own script

- Analysis
- Design the program
- Decompose steps
- Test per step
- Test as a whole

Start with the problem: Analysis

- If you are given the assignment to write a script only a vague description is given. To write a script you need exact code statements. **Analysis** is required to translate the vague description into code statements



Analysis – understand the problem

- Assignment: Get the nucleotide sequence from exon information from delivered gbk files and and make a multifasta file to write the nucleotide sequences to.

- More analysis required, for example
  - how does the gbk file look like?
  - what are the keywords of interest?
  - how can we extract the nucleotide information from exon information
  - what information is given in the exon section and how is the information composed?
  - how does a multifastafile look like?

Design the program

- Once we understand the problem we can translate this to an algorithm:

  From gbk files get the nucleotide sequence from exon information and make a multifasta file to write the nucleotide sequences to.

- For each gbk file
  - Read file
  - Get header info
  - Get dna sequence from exon info
  - Write to multifasta

Decompose steps

- We identified the main steps. We need to decompose a bit more.

- Read the gbk files from the command line
- For each file
  - Get required info (read file)
    - Get version info
    - Get definition info
    - Get DNA sequence info
    - Get exon locations info
  - Compose header and sequence
    - Compose header from VERSION and DEFINITION info
    - Compose exon sequence by extracting dna from dna sequence at exons locations
  - Write to multifasta
    - Write header to multifasta
    - Write exon sequence to multifasta

Design

- Once you understood the problem and you translated this into an algorithm with main steps you can design your body

Design body

```
# IMPORTS
import sys

def read_genbank_file():
    pass

def make_header():
    pass

def get_exon_sequence():
    pass

def make_fastafile():
    pass

def main():
    gbk_files = sys.argv[1:]
    for file in gbk_files:
        read_genbank_file()
        make_header()
        get_dna_sequence()
        make_fastafile()
    print("this is the body")

main()
```

- Read the gbk files from the command line
- For each file
  - Get required info (read file)
    - Get version info
    - Get definition info
    - Get DNA sequence info
    - Get exon locations info
  - Compose header and sequence
    - Compose header from VERSION and DEFINITION info
    - Compose exon sequence by extracting dna from dna sequence at exons locations
  - Write to multifasta
    - Write header to multifasta
    - Write exon sequence to multifasta

Design functions

- For each function think of what goes in, and what should be returned.
- read_genbank file:
  - In: file to read
  - Out: version info, definition info, dna sequence, exon locations
- make_header:
  - In: version info and definition info
  - Out: header

Open and read files

- The read_genbank_file function has multiple things that go out. Here you need to make a strategic decision. If you choose to make separate functions and open and close files you need multiple functions to get the information of interest separately
- If you process the file line by line you need to store information blocks in variables and process them later.
- From a memory perspective it does make more sense to process line by line and not open and close the file all the time.

Design the functions

```
import sys

def read_genbank_file(gbk_file):
    gbk_info = {'definition' : definition,
                'version': version,
                'exons': exon_locations,
                'dna':dna}
    return gbk_info

def make_header(version, definition):
    header = ''
    return header

def get_exon_sequence(dna_sequence, exons_locations):
    exon_sequence = ''
    return exon_sequence

def make_fastafile(header, exon_sequence):
    pass

def main():
    gbk_files = sys.argv[1:]
    for file in gbk_files:
        read_genbank_file(file)
        make_header('', '')
        get_exon_sequence('', '')
        make_fastafile('', '')
    print("this is the body with parameters")

main()
```

You can now expand the body with what goes in (parameters) and what goes out (value you return)

Design function

- Next you do is making a function work. For instance the read_genbank function. First let us decompose the function in smaller steps. We earlier defined the steps
  - Get version info
  - Get definition info
  - Get DNA sequence info
  - Get exon locations info
- Best is to consider each step as a function or process as well. What goes in and what comes out of that process?

Decompose problem

- To get version info we need the VERSION line from position 12
- To get definition info we need DEFINITION line info from position 12
- To get exon position we need to extract the position start and stop from the exon line between the <start:stop>. There are multiple exon lines
- To get dna sequence we need to get the lines dna after the line ORIGIN: till //. There are multiple lines and the data needs to be "cleaned"

Conclusion: we need to read line by line and depending on the content we need to process it.

Code it, test it

```
# IMPORTS
import sys

def read_genbank_file(gbk_file):
    definition = ''
    version = ''
    exon_locations = []
    dna = ''
    f = open(gbk_file)
    for line in f:
        if line.startswith('DEFINITION'):
            definition = line[12:]
        elif line.startswith('VERSION'):
            version = line[12:]
    print(definition, version)
    f.close()
    gbk_info = {'definition' : definition,
                'version': version,
                'exons': exon_locations,
                'dna':dna}
    return gbk_info
```

- Code baby steps and test each step
- In the example on the left first version and definition are extracted from the lines and put in dictionary gbk_inf
- We can test it by printing the version and definition

Code it, test it , use it

- Best is to test it first in the function itself by printing (see previous slide). When it works as expected then use it from the main body and test it from the command line

Code it, test it, use it

```
def make_header(version, definition):
    header = "> " + version.strip() + ' ' + definition.strip() + '\n'
    return header

def main():
    gbk_files = sys.argv[1:]
    for file in gbk_files:
        info = read_genbank_file(file)
        header = make_header(info['definition'],info['version'])
        print(header)
        get_exon_sequence('', '')
        make_fastafile('', '')
    print("this is to test make header")

main()
```

Complex problem?

- Sometimes you do not see immediately how to translate the requirement into an algorithm into code.
- In such a case you simply need to decompose the problem

Complex problem? Decompose!

- To get dna sequence we need to get the lines dna after the line ORIGIN: till //. There are multiple lines and the data needs to be "cleaned"
  - If line startswith ORIGIN: switch 'process on'
  - For each line if 'process on': clean line
    - Strip spaces, returns
    - Make upper
  - If line startswith '//' stop processing

Code it, test it

```
def read_genbank_file(gbk_file):
    definition = ''
    version = ''
    exon_locations = []
    dna = ''
    process = ''
    f = open(gbk_file)
    for line in f:
        if line.startswith('DEFINITION'):
            definition = line[12:]
        elif line.startswith('VERSION'):
            version = line[12:]
        elif line.strip().startswith('ORIGIN'):
            process = 'DNA'
        elif line.startswith('//'):
            process = 'None'
        else:
            if process == 'DNA':
                dna += line[10:].replace(" ", '').replace('\n', '').upper()
    f.close()
    print(dna)
    gbk_info = {'definition' : definition,
                'version': version,
                'exons': exon_locations,
                'dna':dna}
    return gbk_info
```

For each line if 'process on':

- -start from 10
- -Strip spaces, returns
- -Make upper

Test by printing it

Code it, test it, use it

- When it works as expected then use it from the main body and test it from the command line

```
def main():
    gbk_files = sys.argv[1:]
    for file in gbk_files:
        info = read_genbank_file(file)
        header = make_header(info['definition'],info['version'])
        print(info['dna'])
        get_exon_sequence('', '')
        make_fastafile('', '')
    print("this is to test dna")
```

Code it, test it, use it

- Best practice is to start implementing with a simple version. Test and debug it. Use it to ensure that it works as expected. Now, add any features that you want and continue to repeat the Code-Test-Use cycle as many times as required.

- Software is grown, not built

Summary:

- What (Analysis)
- How (Design)
  - Main steps -> body
  - Refine steps with in, out -> parameters & returns
  - Decompose steps in small steps
- Code It (Implementation)
  - Use baby steps
- Test (Testing and Debugging)
  - Test first with printing
- Use (Operation or Deployment)
  - Test it from main body and command line
- Refine