Leiden University
Medical Center

Python Programming

String methods, error and exceptions

Mihai Lefter

Outline

Let's start with a simple GC calculator

seq_toolbox.py

```
1  def calc_gc_percent(seq):
2      at_count, gc_count = 0, 0
3      for char in seq:
4          if char in ('A', 'T'):
5              at_count += 1
6          elif char in ('G', 'C'):
7              gc_count += 1
8
9      return gc_count * 100.0 / (gc_count + at_count)
10
11 print("The sequence 'CAGG' has a %GC of {:.2f}".format(
12           calc_gc_percent("CAGG")))
```

Let's start with a simple GC calculator

seq_toolbox.py

```
1  def calc_gc_percent(seq):
2      at_count, gc_count = 0, 0
3      for char in seq:
4          if char in ('A', 'T'):
5              at_count += 1
6          elif char in ('G', 'C'):
7              gc_count += 1
8
9      return gc_count * 100.0 / (gc_count + at_count)
10
11 print("The sequence 'CAGG' has a %GC of {:.2f}".format(
12           calc_gc_percent("CAGG")))
```

Our script is nice and dandy, but we don't want to edit the source file everytime we calculate a sequence's GC.

The standard library

- A collection of Python modules (or functions, for now) that comes packaged with a default Python installation.
- They're not part of the language per se, more like a batteries included thing.

Our first standard library module: sys

- We'll start by using the simple sys module to make our script more flexible.
- Standard library (and other modules, as we'll see later) can be used via the import statement, for example:

IPython

```
In [1]: import sys
```

- Like other objects so far, we can peek into the documentation of these modules using help, or the IPython ? shortcut. For example:

IPython

```
In [2]: sys?
```

The sys.argv list

- The sys module allows to capture command line arguments with its argv object.
- This is a list of arguments supplied when invoking the current Python session.
- Not really useful for an interpreter session, but very handy for scripts.

IPython

```
In [3]:  sys.argv
Out[3]:  ['/usr/local/bin/ipython']
```

Improving our script with sys.argv

seq_toolbox.py

```
1  import sys
2
3  def calc_gc_percent(seq):
4      at_count, gc_count = 0, 0
5      for char in seq:
6          if char in ('A', 'T'):
7              at_count += 1
8          elif char in ('G', 'C'):
9              gc_count += 1
10
11     return gc_count * 100.0 / (gc_count + at_count)
12
13 input_seq = sys.argv[1]
14 print("The sequence '{}' has a %GC of {:.2f}".format(
15           input_seq, calc_gc_percent(input_seq)))
```

String methods

- Try running the script with `'cagg'` as the input sequence. What happens?
- As we saw earlier, many objects, like those of type `list`, `dict`, or `str`, have useful methods defined on them.
- One way to squash this potential bug is by using Python's string method upper.
- Let's first check out some commonly used string functions.

IPython

```
In [4]: my_str = 'Hello again, ipython!'
In [5]: my_str.upper()
Out[5]: 'HELLO AGAIN, IPYTHON!'

In [6]: my_str.lower()
Out[6]: 'hello again, ipython!'

In [7]: my_str.title()
Out[7]: 'Hello Again, Ipython!'
```

String methods

IPython

```
In [8]:  my_str.startswith('H')
Out[8]:  True

In [9]:  my_str.startswith('h')
Out[9]:  False

In [10]: my_str.split(',')
Out[10]: ['Hello again', ' ipython!']

In [11]: my_str.replace('ipython', 'lumc')
Out[11]: 'Hello again, lumc!'

In [12]: my_str.count('n')
Out[12]: 2
```

Improving our script with upper()

seq_toolbox.py

```
1  import sys
2
3  def calc_gc_percent(seq):
4      at_count, gc_count = 0, 0
5      for char in seq.upper():
6          if char in ('A', 'T'):
7              at_count += 1
8          elif char in ('G', 'C'):
9              gc_count += 1
10
11     return gc_count * 100.0 / (gc_count + at_count)
12
13 input_seq = sys.argv[1]
14 print("The sequence '{}' has a %GC of {:.2f}".format(
15           input_seq, calc_gc_percent(input_seq)))
```

Improving our script with comments and docstrings

seq_toolbox.py

```
1   import sys
2
3   def calc_gc_percent(seq):
4       """
5       Calculates the GC percentage of the given sequence.
6
7       Arguments:
8           - seq - the input sequence (string).
9
10      Returns:
11          - GC percentage (float).
12
13      The returned value is always <= 100.0
14      """
15      at_count, gc_count = 0, 0
16      # Change input to all caps to allow for non-capital
17      # input sequence.
18      for char in seq.upper():
19          if char in ('A', 'T'):
20              at_count += 1
21          elif char in ('G', 'C'):
22              gc_count += 1
23
24      return gc_count * 100.0 / (gc_count + at_count)
25
26  input_seq = sys.argv[1]
27  print("The sequence '{}' has a %GC of {:.2f}".format(
28            input_seq, calc_gc_percent(input_seq)))
```

Errors and exceptions

- Try running the script with `'ACTG123'` as the argument.
  - What happens?
  - Is this acceptable behavior?
- Sometimes we want to put safeguards to handle invalid inputs. In this case we only accept ACTG, all other characters are invalid.
- Python provides a way to break out of the normal execution flow, by raising what's called as an `exception`.
- We can raise exceptions ourselves as well, by using the `raise` statement.

The ValueError built-in exception

- Used on occasions where inappropriate argument values are used, for example when trying to convert the string A to an integer:

IPython

```
In [13]: int('A')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-14-0da6d315d7ad> in <module>()
----> 1 int('A')

ValueError: invalid literal for int() with base 10: 'A'
```

- ValueError is the appropriate exception to raise when your function is called with argument values it cannot handle.

Improving our script by handling invalid inputs

seq_toolbox.py

```
1  def calc_gc_percent(seq):
2      """
3      Calculates the GC percentage of the given sequence.
4
5      Arguments:
6          - seq - the input sequence (string).
7
8      Returns:
9          - GC percentage (float).
10
11     The returned value is always <= 100.0
12     """
13     at_count, gc_count = 0, 0
14     # Change input to all caps to allow for non-capital
15     # input sequence.
16     for char in seq.upper():
17         if char in ('A', 'T'):
18             at_count += 1
19         elif char in ('G', 'C'):
20             gc_count += 1
21         else:
22             raise ValueError('Unexpected character found: {}. Only '
23                              'ACTGs are allowed.'.format(char))
24
25     return gc_count * 100.0 / (gc_count + at_count)
```

Handling corner cases

- Try running the script with `''` as the argument.
  - What happens?
  - Why? Is this a valid input?
- We don't always want to let exceptions stop program flow, sometimes we want to provide alternative flow.
- The `try` ... `except` block allows you to do this.

Improving our script by handling corner cases

seq_toolbox.py

```
1  def calc_gc_percent(seq):
2      """
3      Calculates the GC percentage of the given sequence.
4      ...
5      The returned value is always <= 100.0
6      """
7      at_count, gc_count = 0, 0
8      # Change input to all caps to allow for non-capital
9      # input sequence.
10     for char in seq.upper():
11         if char in ('A', 'T'):
12             at_count += 1
13         elif char in ('G', 'C'):
14             gc_count += 1
15         else:
16             raise ValueError('Unexpected character found: {}. Only '
17                              'ACTGs are allowed.'.format(char))
18
19     # Corner case handling: empty input sequence.
20     try:
21         return gc_count * 100.0 / (gc_count + at_count)
22     except ZeroDivisionError:
23         return 0.0
```

Aim for a minimal try block

- We want to be able to pinpoint the statements that may raise the exceptions so we can tailor our handling.
- Example of code that violates this principle:

```
try:
    my_function()
    my_other_function()
except ValueError:
    my_fallback_function()
```

- A better way would be:

```
try:
    my_function()
except ValueError:
    my_fallback_function()
my_other_function()
```

Be specific when handling exceptions

- The following code is syntactically valid, but never use it:

```
try:
    my_function()
except:
    my_fallback_function()
```

- Always use the full exception name when to make for a much cleaner code:

```
try:
    my_function()
except ValueError:
    my_fallback_function()
except TypeError:
    my_other_fallback_function()
except IndexError:
    my_final_function()
```

Look Before You Leap (LBYL) vs Easier to Ask for Apology (EAFP)

- We could have written our last exception block like so:

```
if gc_count + at_count == 0:
    return 0.0
return gc_count * 100.0 / (gc_count + at_count)
```

- Both approaches are correct and have their own plus and minuses in general.

Leiden University Medical Center

Acknowledgements

Martijn Vermaat
Jeroen Laros
Jonathan Vis