Week 2: Computing in python I, More on writing and testing Python functions

MSc/MRes CMEE 2014-15

Samraat Pawar

Imperial College
London

October 15, 2014

A COUPLE OF THINGS

- Today's submission – any questions? (all code must be functional – use relative paths!)
- Submission due by 5 PM today (unless I told you individually otehrwise)
- Resources on `matlab` - `python` - `R` now available on blackboard

THE MEANING OF LIFE (AND PROGRAMMING)

...some things in life are bad. They can really make you mad. Other things just make you swear and curse. When you're chewing on life's gristle, don't grumble; give a whistle, and this'll help things turn out for the best. And... always look on the bright side of life...

VARIABLE SCOPE, I

- Global variables are visible inside and outside of functions
- Local variables are only accessible inside their function
- Type the following in `scope.py`

```
## Try this first

_a_global = 10

def a_function():
    _a_global = 5
    _a_local = 4
    print "Inside the function, the value is ", _a_global
    print "Inside the function, the value is ",_a_local
    return None

a_function()
print "Outside the function, the value is ", _a_global
```

VARIABLE SCOPE, II

```
17 ## Now try this

19 _a_global = 10

21 def a_function():
       global _a_global
23     _a_global = 5
       _a_local = 4
25     print "Inside the function, the value is ", _a_global
       print "Inside the function, the value is ", _a_local
27     return None

29 a_function()
   print "Outside the function, the value is", _a_global
```

- IF POSSIBLE, AVOID ASSIGNING GLOBALS!

PROFILING IN PYTHON, I

Premature optimization is the root of all evil – Donald Knuth

- Computational speed may not be your initial concern
- Indeed, you should focus on developing clean, reliable, reusable code
- However, speed will become an issue when and if your analysis or modeling becomes complex enough (e.g., food web or large network simulations)
- In that case, knowing which parts of your code take the most time is useful – optimizing those parts may save you lots of time
- To find out what is slowing down your code you need to use "profiling"

PROFILING IN PYTHON, II

- Profiling is easy in IPython – simply type the magic command `%run -p your_function_name`
- Let's write a simple illustrative program and name it `profileme.py`:

```
def a_useless_function(x):
    y = 0
    # eight zeros!
    for i in range(100000000):
        y = y + i
    return 0

def a_less_useless_function(x):
    y = 0
    # five zeros!
    for i in range(100000):
        y = y + i
    return 0

def some_function(x):
    print x
    a_useless_function(x)
    a_less_useless_function(x)
    return 0

some_function(1000)
```

PROFILING IN PYTHON, III

- Now type `%run -p profileme.py`:

```
         54 function calls in 3.652 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.744    2.744    3.648    3.648 profileme.py:1(a_useless_function)
        2    0.905    0.452    0.905    0.452 {range}
        1    0.002    0.002    0.003    0.003 profileme.py:8(a_less_useless_function)
[more output]
```

- The function `range` is taking long – we should use `xrange` instead
- When iterating over a large number of values, `xrange`, unlike `range`, does not create all the values before iteration, but creates them “on demand”
- E.g., `range(1000000)` yields a 4Mb+ list

PROFILING IN PYTHON, IV

- So let's modify the script:

```
1  def a_useless_function(x):
       y = 0
3      # eight zeros!
       for i in xrange(100000000):
5          y = y + i
       return 0
7
   def a_less_useless_function(x):
9      y = 0
       # five zeros!
11     for i in xrange(100000):
           y = y + i
13     return 0

15 def some_function(x):
       print x
17     a_useless_function(x)
       a_less_useless_function(x)
19     return 0

21 some_function(1000)
```

PROFILING IN PYTHON, V

- Again running the magic command `%run -p` yields:

```
         52 function calls in 2.153 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.150    2.150    2.150    2.150 profileme2.py:1(a_useless_function)
        1    0.002    0.002    0.002    0.002 profileme2.py:8(a_less_useless_function)
        1    0.001    0.001    2.153    2.153 {execfile}
[more output]
```

- So we saved 1.499 s! (not enough to grab a pint, but ah well...)

REGULAR EXPRESSIONS IN PYTHON, I

- Regular expressions (regex) are a tool to find patterns in strings, for example,
  - Find DNA motifs in sequence data
  - Navigate through files in a directory
  - Parse text files
  - Extract information from html and xml files
- Regex packages are available for most programming languages (`grep` in UNIX / Linux, where regex first became popular)
- A regex may consist of a combination of “metacharacters” (modifiers) and “regular” or literal characters

REGULAR EXPRESSIONS IN PYTHON, II

- There are 14 metacharacters: [ ] ( ) \ ^ $. | ? * +
- These metacharacters do special things, for example,
  - `[12]` means match target to 1 and if that does not match then match target to 2
  - `[0-9]` means match to any character in range 0 to 9
  - `[^Ff]` means anything except upper or lower case f and `[^a-z]` means everything except lower case a to z
- Everything else is interpreted literally (e.g., `a` is matched by entering `a` in the regex)

REGULAR EXPRESSIONS IN PYTHON, III

- A useful (not exhaustive) list of regex elements is:

`a`	match the character a
`3`	match the number 3
`\n`	match a newline
`\t`	match a tab
`\s`	match a whitespace
`.`	match any character except line break (newline)
`\w`	match any alphanumeric character (including underscore)
`\W`	match any character not covered by `\w` (i.e., match any non-alphanumeric character excl. underscore)
`\d`	match a numeric character
`\D`	match any character not covered by `\d` (i.e., match a non-digit)
`[atgc]`	match any character listed: a, t, g, c
`at|gc`	match at or gc
`[^atgc]`	any character not listed: any character but a, t, g, c
`?`	match the preceding pattern element zero or one times
`*`	match the preceding pattern element zero or more times
`+`	match the preceding pattern element one or more times
`{n}`	match the preceding pattern element exactly n times
`{n,}`	match the preceding pattern element at least n times
`{n,m}`	match the preceding pattern element at least n but not more than m times
`^`	match the beginning of a line
`$`	match the end of a line

REGULAR EXPRESSIONS IN PYTHON, IV

- Regex functions in python are in the module `re` — so we will `import re`
- The simplest python regex function is `re.search`, which searches the string for match to a given pattern — returns a *match object* if a match is found and `None` if not
- **Always** put `r` in front of your regex — it tells python to read the regex in its "raw" (literal) form
- Let's try (type all that follows in `Code/regexs.py`):

```
1  import re
2
3  my_string = "a given string"
4  # find a space in the string
5  match = re.search(r'\s', my_string)
6
7  print match
8  # this should print something like
9  # <_sre.SRE_Match object at 0x93ecdd30>
10
11 # now we can see what has matched
12 match.group()
13
```

REGULAR EXPRESSIONS IN PYTHON, V

```
match = re.search(r's\w*', my_string)

# this should return "string"
match.group()

# NOW AN EXAMPLE OF NO MATCH:
# find a digit in the string
match = re.search(r'\d', my_string)

# this should print "None"
print match
```

- To know whether a pattern was matched, we can use an `if`:

```
str = 'an example'

match = re.search(r'\w*\s', str)

if match:
    print 'found a match:', match.group()
else:
    print 'did not find a match'
```

REGULAR EXPRESSIONS IN PYTHON, VI

```
# Some Basic Examples
match = re.search(r'\d' , "it takes 2 to tango")
print match.group() # print 2

match = re.search(r'\s\w*\s', 'once upon a time')
match.group() # ' upon '

match = re.search(r'\s\w{1,3}\s', 'once upon a time')
match.group() # ' a '

match = re.search(r'\s\w*$', 'once upon a time')
match.group() # ' time'

match = re.search(r'\w*\s\d.*\d', 'take 2 grams of H2O')
match.group() # 'take 2 grams of H2'

match = re.search(r'^\w*.*\s', 'once upon a time')
match.group() # 'once upon a '
## NOTE THAT *, +, and { } are all "greedy":
## They repeat the previous regex token as many times as possible
## As a result, they may match more text than you want

## To make it non-greedy, use ?:
match = re.search(r'^\w*.*?\s', 'once upon a time')
match.group() # 'once '

## To further illustrate greediness, let's try matching an HTML tag:
match = re.search(r'<.+>', 'This is a <EM>first</EM> test')
match.group() # '<EM>first</EM>'
## But we didn't want this: we wanted just <EM>
```

REGULAR EXPRESSIONS IN PYTHON, VII

```
## It's because + is greedy!

## Instead, we can make + "lazy"!
match = re.search(r'<.+?>', 'This is a <EM>first</EM> test')
match.group() # '<EM>'

## OK, moving on from greed and laziness
match = re.search(r'\d*\.?\d*','1432.75+60.22i') #note "\" before "."
match.group() # '1432.75'

match = re.search(r'\d*\.?\d*','1432+60.22i')
match.group() # '1432'

match = re.search(r'[AGTC]+', 'the sequence ATTCGT')
match.group() # 'ATTCGT'

re.search(r'\s+[A-Z]{1}\w+\s\w+', 'The bird-shit frog''s name is Theloderma asper').group() ↩
      # ' Theloderma asper'
## NOTE THAT I DIRECTLY RETURNED THE RESULT BY APPENDING .group()
```

REGULAR EXPRESSIONS IN PYTHON, VIII

REGULAR EXPRESSIONS IN PYTHON, IX

- You can group regexes into meaningful blocks using parentheses:

```
str = 'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and ecological
theory'

# without groups
match = re.search(r"[\w\s]*,\s[\w\.@]*,\s[\w\s&]*",str)

match.group()
'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and ecological theory'

match.group(0)
'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and ecological theory'

# now add groups using ( )
match = re.search(r"([\w\s]*),\s([\w\.@]*),\s([\w\s&]*)",str)

match.group(0)
'Samraat Pawar, s.pawar@imperial.ac.uk, Systems biology and ecological theory'

match.group(1)
'Samraat Pawar'

match.group(2)
's.pawar@imperial.ac.uk'

match.group(3)
'Systems biology and ecological theory'
```

REGULAR EXPRESSIONS IN PYTHON, X

- Some Exercises...
- Translate the following regular expressions into regular English (don't type this in `regexs.py`)!

```
r'^abc[ab]+\s\t\d'

r'^\d{1,2}\/\d{1,2}\/\d{4}$'

r'\s*[a-zA-Z,\s]+\s*'
```

- Write a regex to match dates in format YYYYMMDD, making sure that:
  - Only seemingly valid dates match (i.e., year greater than 1900)
  - First digit in month is either 0 or 1
  - First digit in day ≤ 3

REGULAR EXPRESSIONS IN PYTHON, XI

- Important `re` functions:

`re.compile(reg)`	Compile a regular expression. In this way the pattern is stored for repeated use, improving the speed.
`re.search(reg, text)`	Scan the string and find the first match of the pattern in the string. Returns a `match` object if successful and `None` otherwise.
`re.match(reg, text)`	as `re.search`, but only match the beginning of the string.
`re.split(ref, text)`	Split the text by the occurrence of the pattern described by the regular expression.
`re.findall(ref, text)`	As `re.search`, but return a list of all the matches. If groups are present, return a list of groups.
`re.finditer(ref, text)`	As `re.search`, but return an iterator containing the next match.
`re.sub(ref, repl, text)`	Substitute each non-overlapping occurrence of the match with the text in `repl` (or a function!).

REGULAR EXPRESSIONS IN PYTHON, XII

xkcd.com/208/

READINGS AND RESOURCES PLEASURE

- docs.python.org/2/howto/regex.html
- Google's short class on regex in python: code.google.com/edu/languages/google-python-class/regular-expressions.html
- regular-expressions.info has a good intro, tips and a great array of canned solutions
- codinghorror.com on use and abuse of regex: codinghorror.com/blog/2005/02/regex-use-vs-regex-abuse.html

NO PRACTICAL TODAY, BUT... I

- A good day to get Pracs 0 and 1 sorted out!
- Don't forget to bring today's (functional) scripts under version control: `scope.py, profileme.py, regexs.py`
- Revise regex

NO PRACTICAL TODAY, BUT... II

- And you can start thinking about the following practical (we will revisit it tomorrow):
  - **Objective**: Align two DNA sequences such that they are as similar as possible
  - Start with the longest string and try to position the shorter string in all possible positions
  - For each position, count a “score” : number of bases matched perfectly over the number of bases attempted

NO PRACTICAL TODAY, BUT... III

- Your tasks:
  1. Open and run `../../Practicals/Code/Python/align_seqs.py` and make sure you understand what each line is doing (hint: you can use `pdb` to do this)
  2. Convert `align_seqs.py` to a Python function that takes the DNA sequences as an input from a `.csv` file and saves the best alignment to along with corresponding score in a single text file (hint: remember `pickle`!)
  3. Modify your `align_seqs.py` function such that in alignments with tied number of matches, one with highest proportion of matched pairs is returned. E.g., if alignment 1 matches 5 out of 9 bases, and alignment 2 matches 5 out of 6 bases, alignment 2 is returned.
  4. Make sure you provide a test `.csv` file that the script can call
  5. Extra Credit? Align the `.fasta` sequences from Week 1!