Introduction to

python

for scientific computing

- Lecture 10

Regular Expressions

- A smarter way of searching text
- search&replace
- Relatively advanced topic
    - But incredibly useful
- https://xkcd.com/208/ (https://xkcd.com/208/)

Regular Expressions

- A formal language for defining search patterns
- Enables to search not only for exact strings but controlled variations of that string.
- Why?
- Examples:
    - Find specific patterns in text
        - example: find email in text: `The results should be sent to user@mail.com automatically`
        - Find all hydrocarbons in a text containing compounds `C2H6, NA2, H2O, CH4`
    - American/British spelling, endings and other variants:
        - salpeter, salpetre, saltpeter, nitre, niter or KNO3
        - hemaglobin, heamoglobin, hemaglobins, heamoglobin's
        - catalyze, catalyse, catalyzed...
    - Find/Replace

Regular Expressions

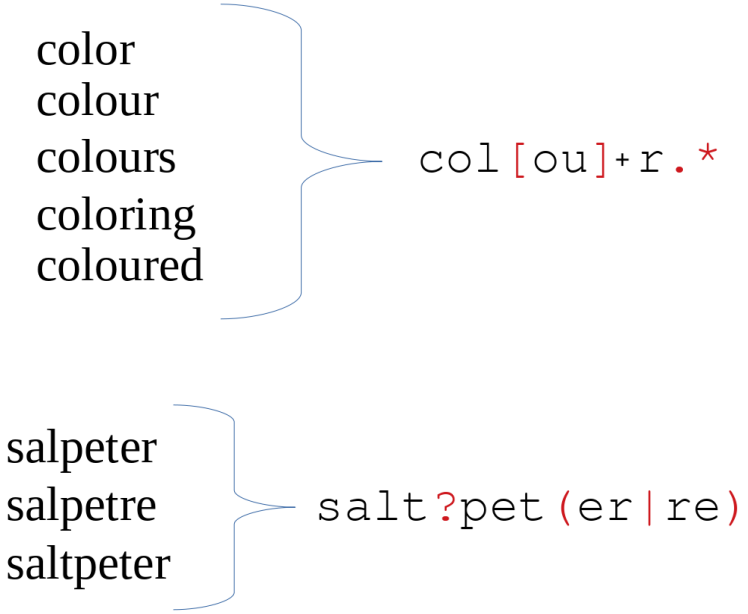
- When?

- To find information
    - in your files
    - in your code
    - in a database
    - online
    - in a bunch of articles
    - ...

- Search/replace
    - becuase → because
    - color → colour
    - `\t` (tab) → " " (four spaces)

- Supported by most programming languages, text editors, search engines...

Defining a search pattern



Common operations

Building blocks for creating patterns

- `.` matches any character (once)
- ? repeat previous pattern 0 or 1 times
- * repeat previous pattern 0 or more times
- + repeat previous pattern 1 or more times

Pattern for matching the colour family

```
colour.*
```

`.*` matches everything (including the empty string)!

Pattern for matching the different spellings

```
salt?peter
```

What about the different endings: er-re?

```
"salt?pet.."
```

saltpeter
"saltpet88"
"salpetin"
"saltpet "

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

`\w+`

```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

`\d+`

```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace

`\s+`

```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

More common operations - classes of characters

- `\w` matches any letter or number, and the underscore
- `\d` matches any digit
- `\D` matches any non-digit
- `\s` matches any whitespace (spaces, tabs, ...)
- `\S` matches any non-whitespace
- `[abc]` matches a single character defined in this set {a, b, c}
- `[^abc]` matches a single character that is **not** a, b or c

`[a-z]` matches all letters between a and z (the english alphabet).

`[a-z]+` matches any (lowercased) english word.

`salt?pet[er]+`

saltpeter
salpetre

~~"saltpet88"~~
~~"salpetin"~~
~~"saltpet "~~

Example - finding patterns in data about genetic mutations

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;DP=190;
GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Each row contains a number of samples, each sample is defined by 0 or 1 separated by /

`0/0 0/1 1/1 ...`

`"[01]/[01]"` (or `"\d/\d"`)

`\s[01]/[01]:`

Example - finding patterns in vcf

```
1   920760  rs80259304  T   C   .   PASS    AA=T;AC=18;AN=120;DP=190;
GP=1:930897;BN=131 GT:DP:CB    0/1:1:SM 0/0:4/SM...
```

- Find all lines containing more than one homozygous sample.

```
... 1/1:...  ... 1/1:...  ...
```

```
.*1/1.*1/1.*
```

```
.*\s1/1:.*\s1/1:.*
```

Test your regexes online before writing the code

- https://regex101.com (https://regex101.com)
- https://regexr.com/ (https://regexr.com/)

Regular expressions in Python

In [1]:
```
import re
```

In [2]:
```
p = re.compile('ab*')
p
```

Out[2]:
```
re.compile(r'ab*', re.UNICODE)
```

Searching

In [11]:
```
p = re.compile('ab.')

if p.search('cdefg e90834uq'):
    print("found")
else:
    print("not found")

result = p.search("abcd")
result
```

```
not found
```

Out[11]:
```
<re.Match object; span=(0, 3), match='abc'>
```

In [35]:
```
print(p.search('cb'))
```

```
None
```

In [12]:
```
p = re.compile('HELLO')
m = p.search('gsdfgsdfgs  HELLO  __!@£§≈[|ÅÄÖ,…’fi]')

print(m)
```

```
<re.Match object; span=(12, 17), match='HELLO'>
```

Case insensitiveness

In [37]:
```
p = re.compile('[a-z]+')
result = p.search('ATGAAA')
print(result)
```

```
None
```

In [38]:
```
p = re.compile('[a-z]+', re.IGNORECASE)

result = p.search('ATGAAA')
result
```

Out[38]:
```
<re.Match object; span=(0, 6), match='ATGAAA'>
```

The match object

In [13]:
```
p = re.compile('[ATCGU]+', re.IGNORECASE)

result = p.search('123 ATGAAA 456')
result
```

Out[13]:
```
<re.Match object; span=(4, 10), match='ATGAAA'>
```

`result.group()` : Return the string matched by the expression

`result.start()` : Return the starting position of the match

`result.end()` : Return the ending position of the match

`result.span()` : Return both (start, end)

In [14]:
```
result.group()
```

Out[14]:
```
'ATGAAA'
```

In [43]:
```
result.start()
```

Out[43]:
```
4
```

In [44]:
```
result.end()
```

Out[44]:
```
10
```

In [45]:
```
result.span()
```

Out[45]:
```
(4, 10)
```

Zero or more...?

In [15]:
```
p = re.compile('.*HELLO.*')
```

In [16]:
```
m = p.search('lots of text  HELLO  more text and characters!!! ^^')
```

In [17]:
```
m.group()
```

Out[17]:
```
'lots of text  HELLO  more text and characters!!! ^^'
```

The `*` is **greedy**.

Finding all the matching patterns

In [18]:
```
p = re.compile('HELLO')
objects = p.finditer('lots of text  HELLO  more text  HELLO ... and character
s!!! ^^')
print(objects)
```

```
<callable_iterator object at 0x7fbd9c0d4e80>
```

In [19]:
```
for m in objects:
    print(f'Found {m.group()} at position {m.start()}')
```

```
Found HELLO at position 14
Found HELLO at position 32
```

In [51]:
```
objects = p.finditer('lots of text  HELLO  more text  HELLO ... and character
s!!! ^^')
for m in objects:
    print('Found {} at position {}'.format(m.group(), m.start()))
```

```
Found HELLO at position 14
Found HELLO at position 32
```

How to find a full stop?

In [20]:
```
txt = "The first full stop is here: ."
p = re.compile('.')

m = p.search(txt)
print('"{}" at position {}'.format(m.group(), m.start()))
```

```
"T" at position 0
```

In [21]:
```
p = re.compile('\.')

m = p.search(txt)
print('"{}" at position {}'.format(m.group(), m.start()))
```

```
"." at position 29
```

More operations

- \ escaping a character
- ^ beginning of the string
- $ end of string
- | boolean or

```
^hello$
```

```
salt?pet(er|re) | nit(er|re) | KNO3
```

Substitution

Finally, we can fix our spelling mistakes!

In [54]:
```
txt = "Do it   becuase   I say so,     not becuase you want!"
```

In [55]:
```
import re
p = re.compile('becuase')
txt = p.sub('because', txt)
print(txt)
```

```
Do it   because   I say so,     not because you want!
```

In [56]:
```
p = re.compile('\s+')
p.sub(' ', txt)
```

Out[56]:
```
'Do it because I say so, not because you want!'
```

Overview

- Construct regular expressions

```
p = re.compile()
```

- Searching

```
p.search(text)
```

- Substitution

```
p.sub(replacement, text)
```

Typical code structure:

```
p = re.compile( ... )
m = p.search('string goes here')
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

Regular expressions

- A powerful tool to search and modify text
- There is much more to read in the docs (https://docs.python.org/3/library/re.html)
- Note: regex comes in different flavours. If you use it outside Python, there might be small variations in the syntax.

Sum up!

Processing files - looping through the lines

```
fh = open('myfile.txt')
for line in fh:
    do_stuff(line)
```

Store values

```
iterations = 0
information = []

fh = open('myfile.txt', 'r')
for line in fh:
    iterations += 1
    information += do_stuff(line)
```

Values

- Base types:

```
- str     "hello"
- int     5
- float   5.2
- bool    True
```

- Collections:

```
- list  ["a", "b", "c"]
- dict  {"a": "alligator", "b": "bear", "c": "cat"}
- tuple ("this", "that")
- set   {"drama", "sci-fi"}
```

Assign values

```
iterations = 0
score = 5.2
```

Compare and membership

```
+, -, *,...  # mathematical
and, or, not  # logical
==, !=        # comparisons
<, >, <=, >=  # comparisons
in            # membership
```

```
value = 4
nextvalue = 1
nextvalue += value
print('nextvalue: ', nextvalue, 'value: ', value)
```

```
nextvalue:  5 value:  4
```

In [23]:

```
x = 5
y = 7
z = 2
x > 6 and y == 7 or z > 1
```

Out[23]:

```
True
```

In [25]:

```
(x > 6 and y == 7) |z > 1
```

Out[25]:

```
True
```

Strings

- Works like a list of characters
    - `s += "more words"  # add content`
    - `s[4]  # get character at index 4`
    - `'e' in s  # check for membership`
    - `len(s)  # check size`

- But are immutable
    - `> s[2] = 'i'`

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Strings

Raw text

- Common manipulations:
  - `s.strip()  # remove unwanted spacing`
  - `s.split()  # split line into columns`
  - `s.upper(), s.lower()  # change the case`
- Regular expressions help you find and replace strings.
  - ```
    p = re.compile('A.A.A')
    p.search(dnastring)
    ```
  - ```
    p = re.compile('T')
    p.sub('U', dnastring)
    ```

```
import re

p = re.compile('p.*\sp')  # the greedy star!

p.search('a python programmer writes python code').group()
```

Out[26]:
```
'python programmer writes p'
```

Collections

Can contain strings, integer, booleans...

- **Mutable**: you can *add*, *remove*, *change* values
    - Lists:

      ```
      mylist.append('value')
      ```

    - Dicts:

      ```
      mydict['key'] = 'value'
      ```

    - Sets:

      ```
      myset.add('value')
      ```

Collections

- Test for membership:

```
value in myobj
```

- Check size:

```
len(myobj)
```

Lists

- Ordered!

```
todolist = ["work", "sleep", "eat", "work"]

todolist.sort()
todolist.reverse()
todolist[2]
todolist[-1]
todolist[2:6]
```

```
todolist = ["work", "sleep", "eat", "work"]
```

In [28]:
```
todolist.sort()
print(todolist)
```

```
['eat', 'sleep', 'work', 'work']
```

In [29]:
```
todolist.reverse()
print(todolist)
```

```
['work', 'work', 'sleep', 'eat']
```

In [64]:
```
todolist[2]
```

Out[64]:
```
'sleep'
```

In [65]:
```
todolist[-1]
```

Out[65]:
```
'eat'
```

In [66]:
```
todolist[:]
```

Out[66]:
```
['sleep', 'eat']
```

Dictionaries

- Keys have values

```
mydict = {"a": "alligator", "b": "bear", "c": "cat"}
counter = {"cats": 55, "dogs": 8}

mydict["a"]
mydict.keys()
mydict.values()
```

```
counter = {'cats': 0, 'others': 0}

for animal in ['zebra', 'cat', 'dog', 'cat']:
    if animal == 'cat':
        counter['cats'] += 1
    else:
        counter['others'] += 1
    if "zebra" in counter:
        counter["zebra"] +=1
    else:
        counter["zebra"] = 1

counter
```

Out[31]: {'cats': 2, 'others': 2, 'zebra': 4}

Sets

- Bag of values
    - No order
    - No duplicates
    - Fast membership checks
    - Logical set operations (union, difference, intersection...)

```
myset = {"drama", "sci-fi"}

myset.add("comedy")

myset.remove("drama")
```

```
todolist = ["work", "sleep", "eat", "work"]

todo_items = set(todolist)
todo_items
```

Out[69]: {'eat', 'sleep', 'work'}

In [71]:
```
todo_items.add("study")
todo_items
```

Out[71]: {'eat', 'sleep', 'study', 'work'}

In [72]:
```
todo_items.add("eat")
todo_items
```

Out[72]: {'eat', 'sleep', 'study', 'work'}

Tuples

- A group (usually two) of values that belong together
    - `tup = (max_length, sequence)`
    - An ordered sequence (like lists)
    - `length = tup[0]  # get content at index 0`
    - Immutable

In [32]:
```
tup = (2, 'xy')
tup[0]
```

Out[32]:
```
2
```

In [33]:
```
tup[0] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [33], in <cell line: 1>()
----> 1 tup[0] = 2

TypeError: 'tuple' object does not support item assignment
```

Tuples in functions

```
def find_longest_seq(file):
    # some code here...
    return length, sequence


answer = find_longest_seq(filepath)
print('length', answer[0])
print('sequence', answer[1])


answer = find_longest_seq(filepath)
length, sequence = find_longest_seq(filepath)
```

Deciding what to do

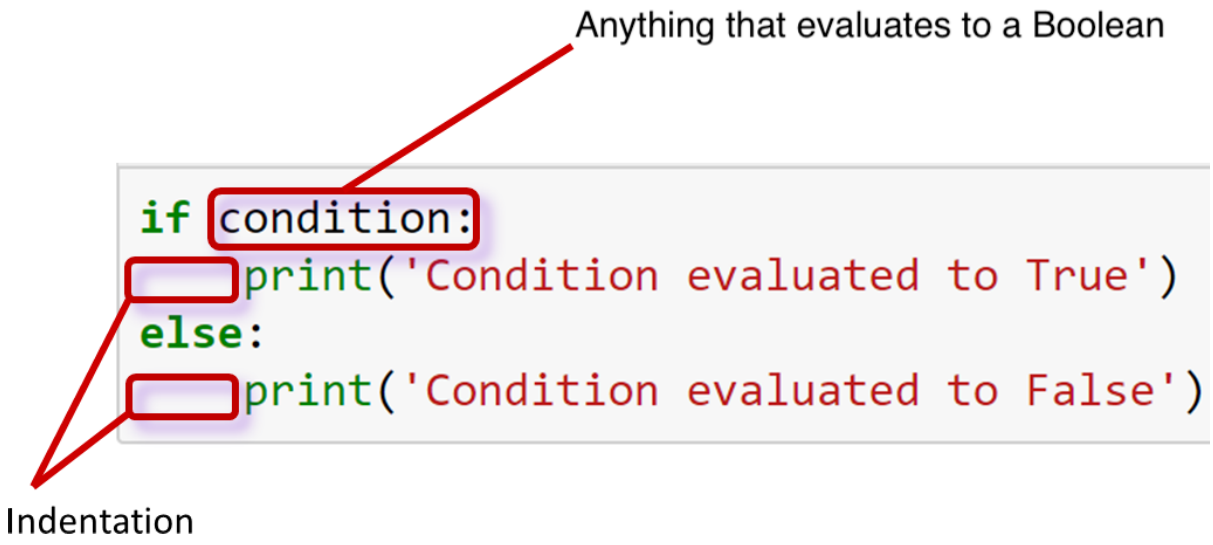
```
if count > 10:
    print('big')
elif count > 5:
    print('medium')
else:
    print('small')
```

```
shopping_list = ['bread', 'egg', ' butter', 'milk']
tired         = True

if len(shopping_list) > 4:
    print('Really need to go shopping!')
elif not tired:
    print('Not tired? Then go shopping!')
else:
    print('Better to stay at home')
```

```
Better to stay at home
```

Deciding what to do - if statement



Program flow - for loops

```
information = []
fh = open('myfile.txt', 'r')

for line in fh:
    if is_comment(line):
       use_comment(line)
    else:
       information = read_data(line)
```

```
for line in open('myfile.txt', 'r'):
    if is_comment(line):
        use_comment(line)
    else:
        information = read_data(line)
```

Program flow - while loops

```
keep_going = True
information = []
index = 0

while keep_going:
    current_line = lines[index]
    information += read_line(current_line)
    index += 1
    if check_something(current_line):
        keep_going = False
```

```
while keep_going:
    current_line = lines[index]
    information += read_line(current_line)
    index += 1
    if check_someting(current_line):
        keep_going = False
```

Different types of loops

`For` loop

is a control flow statement that performs operations over a known amount of steps.

`While` loop

is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.

Which one to use?

`For` loops - standard for iterations over lists and other iterable objects

`While` loops - more flexible and can iterate an unspecified number of times

```
user_input = "thank god it's friday"
for letter in user_input:
    print(letter.upper())
```

```
T
H
A
N
K

G
O
D

I
T
'
S

F
R
I
D
A
Y
```

In [37]:

```
i = 0
while i < len(user_input):
    letter = user_input[i]
    print(letter.upper())
    i += 1
```

THANK

GOD

IT'S

FR

Controlling loops

- `break` - stop the loop
- `continue` - go on to the next iteration

```python
user_input = "thank god it's friday"
for letter in user_input:

    if letter == 'd':
        continue
    print(letter.upper())
```

```
T
H
A
N
K

G
O

I
T
'
S

F
R
I
A
Y
```

Watch out!

In [79]:
```
# DON'T RUN THIS
i = 0
while i > 10:
    print(user_input[i])
```

While loops may be infinite!

Input/Output

- In:
    - Read files: `fh = open(filename, 'r')`
        - `for line in fh:`
            - `fh.read()`
            - `fh.readlines()`
    - Read information from command line: `sys.argv[1:]`
- Out:
    - Write files: `fh = open(filename, 'w')`
        - `fh.write(text)`
    - Printing: `print('my_information')`

Input/Output

- Open files should be closed:
    - `fh.close()`

Code structure

- Functions
- Modules

Functions

- A named piece of code that performs a certain task.

```
def functionName(arg1, arg2, arg3):

    finalValue = 0

    # Here is some code where you can do
    # calculations etc, on arg1, arg2, arg3
    # and update finalValue

    return FinalValue
```

- Is given a number of input arguments
    - to be used (are in scope) within the function body
- Returns a result (maybe `None`)

Functions - keyword arguments

```
def prettyprinter(name, value, delim=":", end=None):
    out = "The " + name + " is " + delim + " " + value
    if end:
        out += end
    return out
```

- used to set default values (often `None`)
- can be skipped in function calls
- improve readability

Using your code

Any longer pieces of code that have been used and will be re-used should be saved

- Save it as a file `.py`
- To run it: `python3 mycode.py`
- Import it: `import mycode`

Documentation and comments

- `""" This is a doc-string explaining what the purpose of this function/module is """`
- `# This is a comment that helps understanding the code`

- Comments *will* help you
- Undocumented code rarely gets used
- Try to keep your code readable: use informative variable and function names

```
import sys
import re
import argparse


def mkParser():
  parser = argparse.ArgumentParser(description = "Calculates allele frequency and depth for each variant in a vcf file")
  parser.add_argument("--vcf",          type = str,    required = True,   help="a file in vcf format")
  parser.add_argument("--out",          type = str,    required = True,   help="the name of the output file")

  return parser.parse_args()


def count_variants(infile, out):
  out = open(out,"w")
  out.write('variant\taverage_total_depth_over_variants\tno_samples\tfrequency\n')
  for line in infile:
    if not line.startswith('#'):
      linecol = line.strip().split('\t')
      i = 0
      alt = linecol[4].split(',')
      while i < len(alt):
        out.write(linecol[0]+'_'+linecol[1]+'_'+linecol[3]+'_'+str(alt[i])+'\t')
        j = 9
        count_hom   = 0
        count_het   = 0
        samples     = 0
        depth       = 0
        while j < len(linecol):
          cols = linecol[j].split(':')
          if cols[0] != './.' and cols[0] != '.' and cols[2] != '.':
            samples += 1
            if cols[0] == '0/'+str(i+1) or cols[0] == str(i+1)+'/0':
              depth += int(cols[2])
              count_het += 1
            elif cols[0] == str(i+1)+'/'+str(i+1):
              depth += int(cols[2])
              count_hom += 1
          j += 1

        if samples > 0 and count_het+count_hom > 0:
```

```
        if samples != 0 and count_het+count_hom != 0:
          freq = (count_het+(2*count_hom))/(samples*2)
          depth_av = depth/(count_het+count_hom)
        else:
          freq = 'missing'
          depth_av = 'missing'
        out.write(str(depth_av)+'\t'+str(samples)+'\t'+str(freq)+'\n')
        i += 1

out.close()
```

Why programming?

- Computers are fast
- Computers don't get bored
- Computers don't get sloppy

- Create reproducable results
- Extract large amount of information

Final advice

- Stop and think before you start coding
    - use pseudocode
    - use top-down programming (divide and conquer)
    - use paper and pen
    - take breaks

- You know the basics - don't be afraid to try, it's the only way to learn
- You will get faster

Final advice (for real)

- Getting help
    - search the web ("pandas filter dataframe multiple columns", "python find all regexes")
    - ask colleauges
    - talk about your problem (get a rubber duck https://en.wikipedia.org/wiki/Rubber_duck_debugging (https://en.wikipedia.org/wiki/Rubber_duck_debugging))
    - maybe send me an email

Final project

- Just a way to show you understand the basics
- Nothing complicated if you have gone through the slides
- Instructions here (https://github.com/clami66/workshop-python/blob/0422/project/instructions.ipynb) (download link) (https://github.com/clami66/workshop-python/raw/0422/project/instructions.ipynb)