Introduction to

python

with Application to Bioinformatics

- Day 5

Sharing code

HackMD

- Pair programming section

Share code snippet

- **Pastebin**

Collaboration space for notebooks

- **Colab**

Share codebase - Advanced option

- **Github**

Review

- Lists
  - Create a list named letters_list containing the elements `'a'`, `'b'`, `'c'`.
  - Reverse the list letters_list
- Dictionaries
  - Create a dictionary called letters_dict containing the keys `a` and `b`. Both should have the value 1.
  - Change the value of `a` to 2.
- Formatting
  - Set the variable `title` to `"A movie"`
  - Set the variable `rating` to `10`.
  - Use formatting to produce the following string: `"The movie A movie got rating 10!"`

In [17]:

```
# Create a list containing the elements `'a'`, `'b'`, `'c'`
```

In [16]:

```
# Reverse it
```

In [15]:

```
# Create a dictionary containing the keys a and b. Both should have the value 1
```

In [14]:

```
# Change the value of a to 2
```

In [13]:

```
# Set the variable `title` to `"A movie"` and `rating` to 10.
```

In [12]:

```
# Use formatting to produce: "The movie A movie got rating 10!"
```

TODAY

- review
- regex
- sumup

Review Day 4

- More control!
    - variables scope
    - `None`
    - keyword arguments
    - documentation, comments...
- Pandas

In [18]:

```
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    return ['Function element 1', 'Function element 2']

my_list = function_returning_values()
print(my_list)
```

```
['Function element 1', 'Function element 2']
```

In [19]:

```
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    my_list = ['Function element 1', 'Function element 2']

function_returning_values()
print(my_list)
```

```
['Initial element 1', 'Initial element 2']
```

In [20]:

```
my_list = ['Initial element 1', 'Initial element 2']

def function_returning_values():
    my_list = ['Function element 1', 'Function element 2']
my_list = function_returning_values()
print(my_list)
```

```
None
```

In [21]:

```
# `None` means "nothing". Use it to check your variables

variable = None
if variable:
    print('if variable')
if not variable:
    print('if not variable')
if variable is not None:
    print('if variable is not None')
if variable is None:
    print('if variable is None')
```

```
if not variable
if variable is None
```

KEYWORD ARGUMENTS

```
open(filename, encoding="utf-8")
```

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

DOCUMENTATION AND GETTING HELP

- `help(sys)`

- write comments `# why do I do this?`
- write documentation `"""what is this? how do you use it?"""`

WRITING READABLE CODE

```
def f(a, b):
    for c in open(a):
        if c.startswith(b):
            print(c)
```

==>

```
def print_lines(filename, start):
    """Print all lines in the file that starts with the given string."""
    for line in open(filename):
        if line.startswith(start):
            print(line)
```

Care about the names of your variables and functions

Pandas

- Read tables

```
dataframe = pandas.read_table('mydata.txt', sep='|', index_col=0)
dataframe = pandas.read_csv('mydata.csv')
```

- Select rows and colums

```
dataframe.columname
dataframe.loc[rowname]
dataframe.loc[dataframe.age == 20 ]
```

- Plot it

```
dataframe.plot(kind='line', x='column1', y='column2')
```

TODAY

- Regular expressions
- Sum up of the course

Regular Expressions

- **A smarter way of searching text**
- **search&replace**
- **Relatively advanced topic**

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 variations in a protein or DNA sequence
        - `"MVR???A"`
        - `"ATG???TAG"`
    - American/British spelling, endings and other variants:
        - salpeter, salpetre, saltpeter, nitre, niter or KNO3
        - hemaglobin, heamoglobin, hemaglobins,

    - heamoglobin's
    - catalyze, catalyse, catalyzed...
- A pattern in a vcf file
    - a digit appearing after a tab

Regular Expressions

- When?

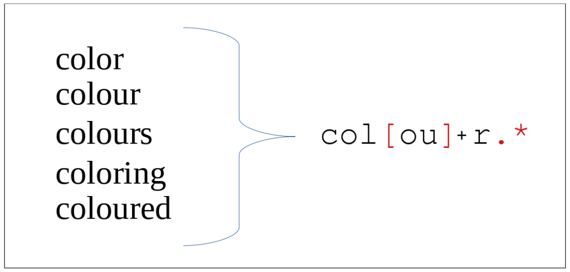
- To find information
  - in your `vcf` or `fasta` files
  - in your code
  - in your next essay
  - in a database
  - online
  - in a bunch of articles
  - ...

- Search/replace
  - becuase → because
  - color → colour

    - `\t` (tab) → `"    "` (four spaces)

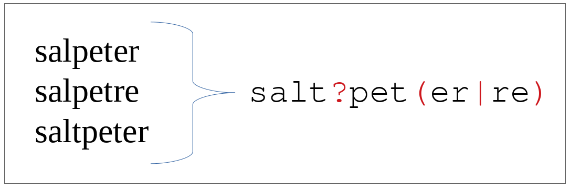
- Supported by most programming languages, text editors, search engines...

Defining a search pattern



color
colour
colours
coloring
coloured

`col[ou]+r.*`



salpeter
salpetre
saltpeter

`salt?pet(er|re)`

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 "

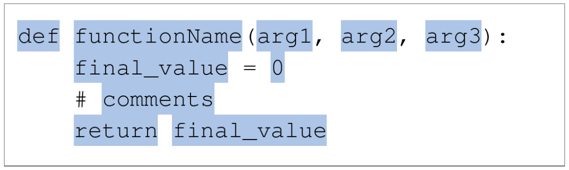
MORE COMMON OPERATIONS - CLASSES OF CHARACTERS

- `\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+`

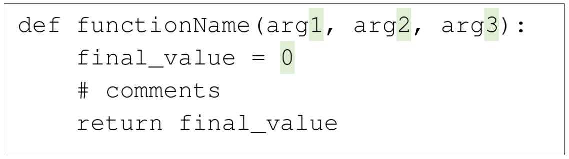


```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

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+`

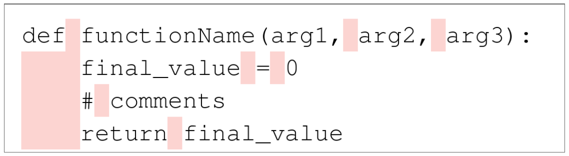


```
def functionName(arg1, arg2, arg3):
    final_value = 0
    # comments
    return final_value
```

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 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 a sample:

`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:.*
```

Exercise 1

- `.` 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
- `\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 any (lowercased) letter from the english alphabet

- `.*` matches anything
- **https://regexr.com/**

→ Notebook Day_5_Exercise_1 (~30 minutes)

Regular expressions in Python

In [22]:

```
import re
```

In [23]:

```
p = re.compile('ab*')
p
```

Out[23]:

```
re.compile(r'ab*', re.UNICODE)
```

Searching

In [34]:

```
p = re.compile('ab*')

p.search('abc')
```

Out[34]:

```
<re.Match object; span=(0, 2), match='ab'>
```

In [35]:

```
print(p.search('cb'))
```

```
None
```

In [36]:

```
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 [41]:

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

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

Out[41]:

```
<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 [42]:

```
result.group()
```

Out[42]:

```
'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 [46]:

```
p = re.compile('.*HELLO.*')
```

In [47]:

```
m = p.search('lots of text  HELLO  more text and characters!!! ^^')
```

In [48]:

```
m.group()
```

Out[48]:

```
'lots of text  HELLO  more text and characters!!! ^^'
```

The * is **greedy**.

Finding all the matching patterns

In [49]:

```
p = re.compile('HELLO')
objects = p.finditer('lots of text  HELLO  more text  HELLO ... and characters!!! ^^')
print(objects)
```

```
<callable_iterator object at 0x7fc79ccc02e0>
```

In [50]:

```
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 characters!!! ^^')
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 [52]:

```
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 [53]:

```
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**
- Note: regex comes in different flavours. If you use it outside Python, there might be small variations in the syntax.

Exercise 2

- `.` 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
- `\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 any (lowercased) letter from the english alphabet

- `.*` matches anything
- `\` escaping a character
- `^` beginning of the string
- `$` end of string
- `|` boolean `or`

Read more: full documentation **https://docs.python.org/3.6/library/re.html**

→ Notebook Day_5_Exercise_2 (~30 minutes)

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
```

In [57]:

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

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

In [58]:

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

Out[58]:

```
True
```

In [59]:

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

Out[59]:

```
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)
      ```

In [60]:

```
import re

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

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

Out[60]:

```
'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]
```

In [61]:

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

In [62]:

```
todolist.sort()
print(todolist)
```

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

In [63]:

```
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[2:]
```

Out[66]:

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

DICTIONARIES

- Keys have values

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

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

In [67]:

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

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

counter
```

Out[67]:

```
{'cats': 2, 'others': 2}
```

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")
```

In [69]:

```
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 [73]:

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

Out[73]:

```
2
```

In [74]:

```
tup[0] = 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-74-874559a0c62a> in <module>
----> 1 tup[0] = 2

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

TUPLES IN FUNCTIONS

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

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

```python
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')
```

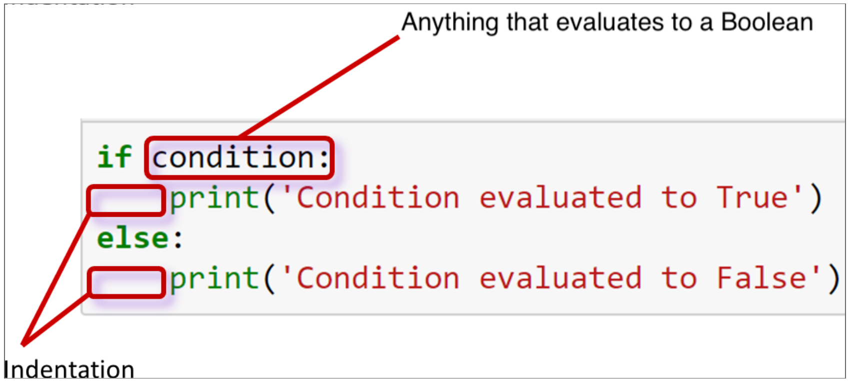
In [75]:

```
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



Anything that evaluates to a Boolean

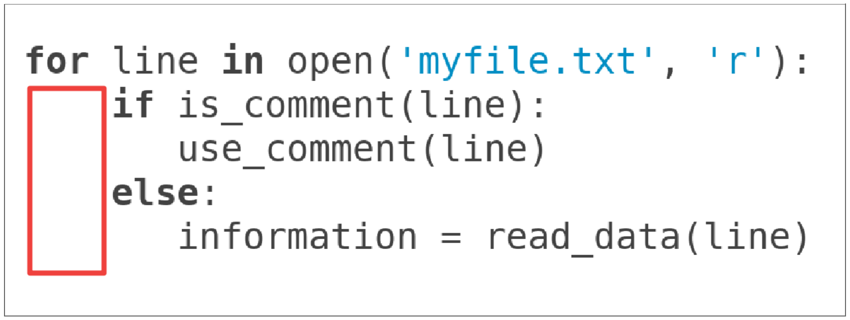
```python
if condition:
    print('Condition evaluated to True')
else:
    print('Condition evaluated to False')
```

Indentation

PROGRAM FLOW - FOR LOOPS

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

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

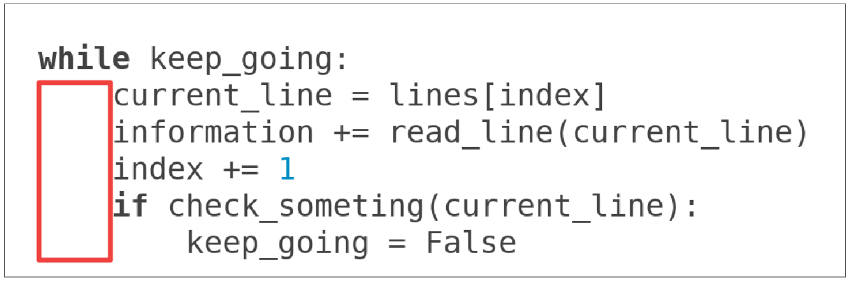


```
for line in open('myfile.txt', 'r'):
    if is_comment(line):
        use_comment(line)
    else:
        information = read_data(line)
```

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

In [76]:

```
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 [77]:

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

```
T
H
A
N
K
 
G
O
D
 
I
T
'
S
 
F
R
I
D
A
Y
```

CONTROLLING LOOPS

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

In [78]:

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

```
T
H
A
N
K
 
G
O
D
```

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

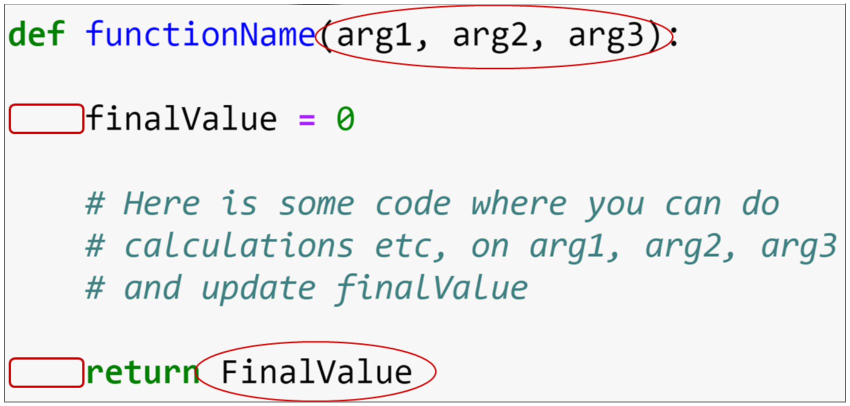
- Open files should be closed:
    - `fh.close()`

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

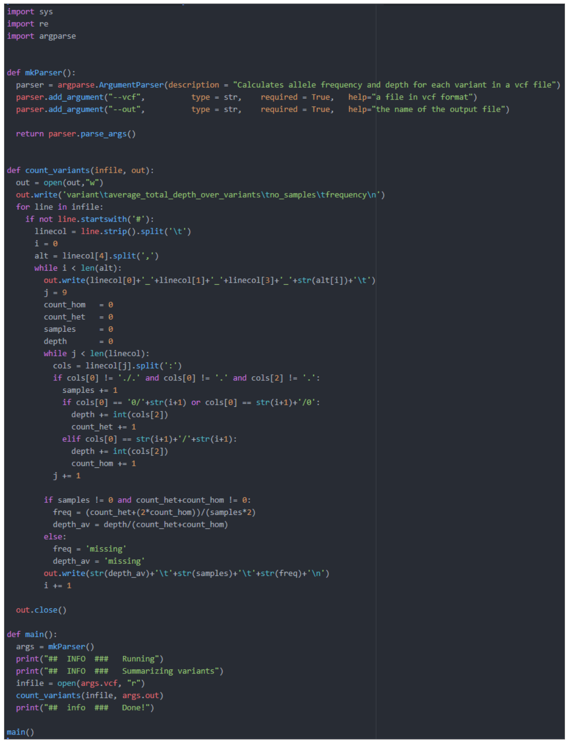
Any longer pieces of code that have been used and will be re-used should be saved

- 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



```python
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:
          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()

def main():
  args = mkParser()
  print("##  INFO  ###   Running")
  print("##  INFO  ###   Summarizing variants")
  infile = open(args.vcf, "r")
  count_variants(infile, args.out)
  print("##  info  ###   Done!")

main()
```

WHY PROGRAMMING?

Endless possibilities!

- reverse complement DNA
- custom filtering of VCF files
- plotting of results
- all excel stuff!

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
    - 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
    - ask colleauges
    - talk about your problem (get a rubber duck **https://en.wikipedia.org/wiki/Rubber_duck_debugging**)
    - search the web
    - NBIS drop-ins

Now you know Python!

🎉

Well done!