Introduction to

python

with Application to Bioinformatics

- Day 5

Review

- Dictionaries
    - Create a dictionary containing the keys `a` and `b`. Both should have the value 1.
    - Change the value of `b` to 5.
- Lists
    - Create a list containing the elements `'a'`, `'b'`, `'c'`.
    - Reverse it
- Set the variable `title` to `"A movie"` and `rating` to `10`.
    - Use formatting to produce the following string:

      `"The movie the movie got rating 10!"`

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

In [ ]:
```
# Change the value of b to 5
```

In [ ]:
```
# Create a list containing the elements `'a'`, `'b'`, `'c'`
```

In [ ]:
```
# Reverse it
```

In [ ]:
```
# Set the variable `title` to `"A movie"` and `rating` to 10.
```

In [ ]:
```
# Use formatting to produce: "The movie the movie got rating 10!"
```

TODAY

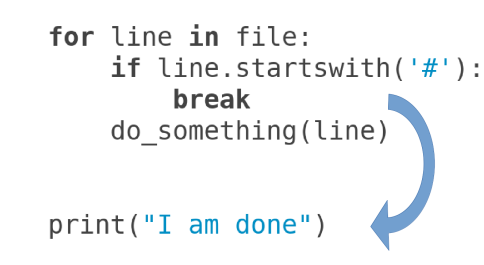
- review
- regex
- sumup

Review Day 4

- More control!
    - `None`
    - `break`, `continue`
    - keyword arguments
    - documentation, comments...
- Pandas

Control loops

- `break` a loop => stop it



Control loops

- `continue` => go on to the next iteration

```
for line in file:
    if line.startswith('#'):
        continue
    do_something(line)


print("I am done")
```

None

`None` means "nothing". Is neither true nor false.

```
if value:
   print('value is not 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)`

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

Writing readable code

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

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)
```

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**

Regular Expressions

Regular Expressions

- A formal language for defining search patterns

Regular Expressions

- A formal language for defining search patterns
- Let's you search not only for exact strings but controlled variations of that string.

Regular Expressions

- A formal language for defining search patterns
- Let's you search not only for exact strings but controlled variations of that string.
- Why?

Regular Expressions

- A formal language for defining search patterns
- Let's you 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

Regular Expressions

- When?

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
    - ...

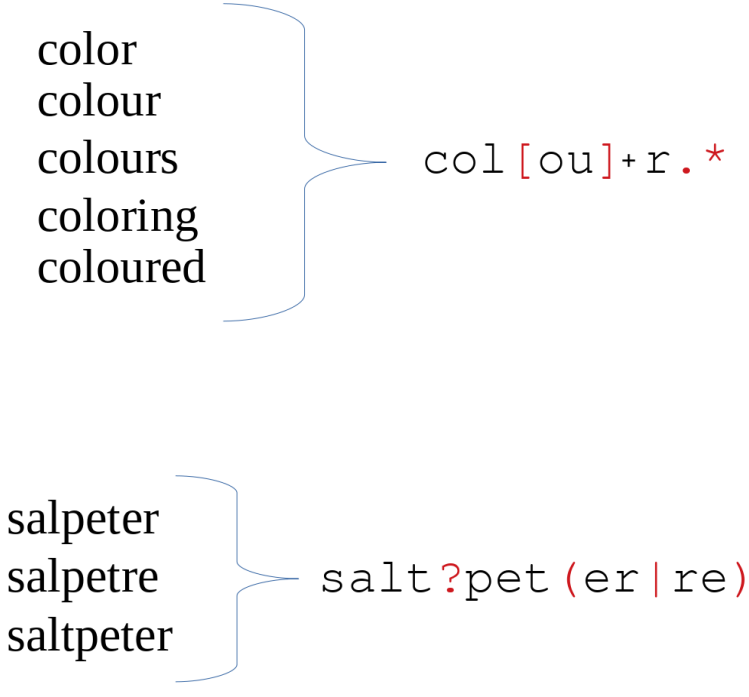
Regular Expressions

- 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)

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



Common operations

- `.` 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

```
colour.*
```

```
salt?peter
```

Common operations

- `.` 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

`colour.*`

`salt?peter`

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

Common operations

- `.` 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

`colour.*`

`salt?peter`

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

`"salt?pet.."`

Common operations

- `.` 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

`colour.*`

`salt?peter`

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

`"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

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.

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...
```

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` ...

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"`)

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.

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:...  ...
```

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.*
```

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

→ Notebook Day_5_Exercise_1 (~30 minutes)

Regular expressions in Python

Regular expressions in Python

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

Regular expressions in Python

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

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

Searching

Searching

In [ ]:

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

p.search('abc')
```

Searching

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

p.search('abc')
```

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

Searching

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

p.search('abc')
```

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

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

print(m)
```

Case insensitiveness

In [ ]:

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

Case insensitiveness

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

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

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

The match object

The match object

In [ ]:

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

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

The match object

In [ ]:

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

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

`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)

The match object

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

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

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

The match object

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

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

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

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

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

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

Zero or more...?

In [ ]:

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

Zero or more...?

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

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

Zero or more...?

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

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

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

Zero or more...?

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

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

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

The `*` is **greedy**.

Finding all the matching patterns

In [ ]:

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

Finding all the matching patterns

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

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

Finding all the matching patterns

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

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

In [ ]:
```
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()))
```

How to find a full stop?

In [ ]:

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

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

How to find a full stop?

In [ ]:

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

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

In [ ]:

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

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

More operations

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

More operations

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

```
^hello$
```

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

Substitution

Finally, we can fix our spelling mistakes!

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

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

Substitution

Finally, we can fix our spelling mistakes!

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

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

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

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

```
for line in open('myfile.txt', 'r'):
    do_stuff(line)
```

Store values

```
iterations = 0
information = []

for line in open('myfile.txt', 'r'):
    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
```

Modify values and compare

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

In [ ]:

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

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`

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`

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 [ ]:

```
import re

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

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

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

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

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

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

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

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

Dictionaries

- Keys have values

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

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

In [ ]:

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

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

counter
```

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")
```

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")
```

for m in objects: print(f'Found {m.group()} at position {m.start()}')

In [ ]:

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

todo_items = set(todolist)
todo_items
```

In [ ]:

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

todo_items = set(todolist)
todo_items
```

In [ ]:

```
todo_items.add("study")
todo_items
```

In [ ]:

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

todo_items = set(todolist)
todo_items
```

In [ ]:

```
todo_items.add("study")
todo_items
```

In [ ]:

```
todo_items.add("eat")
todo_items
```

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

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

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

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

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


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

```
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

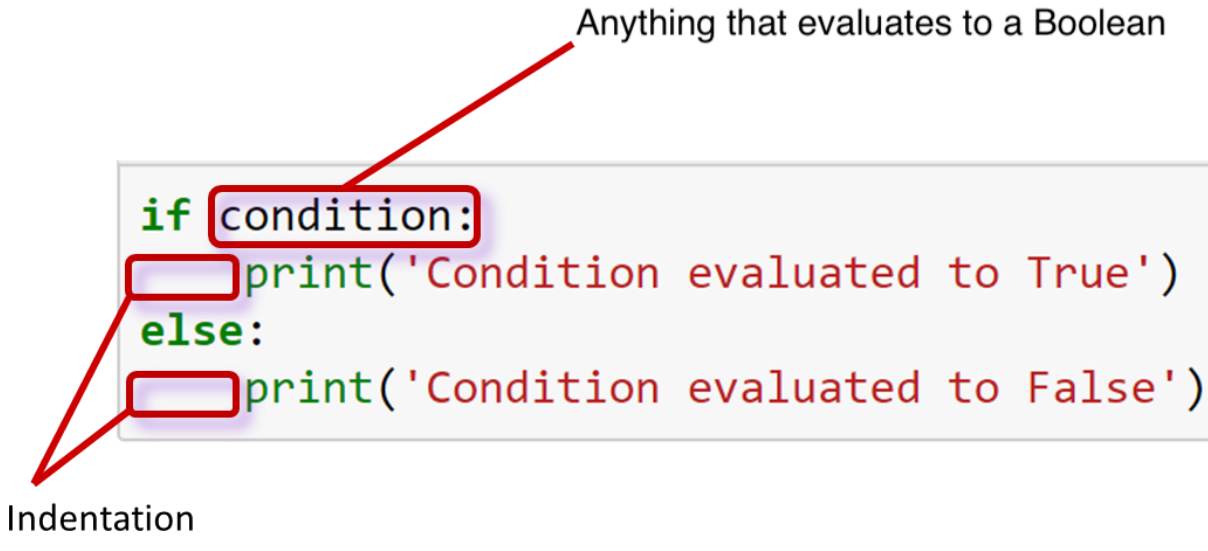
```
if count > 10:
    print('big')
elif count > 5:
    print('medium')
else:
    print('small')
```

In [ ]:

```
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')
```

Deciding what to do - if statement



Program flow - for loops

```
information = []

for line in open('myfile.txt', 'r'):
    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 [ ]:

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

In [ ]:

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

Controlling loops

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

In [ ]:

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

Watch out!

In [ ]:

```
#Don't run this!
i = 0
while i > 10:
    print(user_input[i])
```

Watch out!

In [ ]:

```
#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`

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

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

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:
        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?

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

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

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!