Introduction to

with Application to Bioinformatics

- DAY 5

Day 5

- **Session 1**
    - Quiz: Review of Day 4
    - Lecture: Go through questions
    - Lecture: Introduction to regex
    - Ex1: Find the pattern using regex
- **Session 2**
    - Lecture: Regex in Python
    - Ex2: Regex using Python
    - PyQuiz 5.1
- **Session 3**
    - Lecture: Sum up
    - Ex3: Final exercise
- **Project time**

Quiz: Review Day 4

Go to Canvas, `Modules -> Day 5 -> Review Day 4`
~15 minutes

1. What happens if you declare a variable with the same name inside and outside a function?

The variable inside the function has a separate scope and does not affect the one outside

```
In [ ]: name = "Max"
        def changeName():
    name = "Niko"
    print(f"name inside the function: {name}")
changeName()
print(f"name outside of the function: {name}")
```

2. What is the difference between positional arguments and keyword arguments?

Keyword arguments can be given in any order, while positional arguments depend on the function's order

```
In [ ]: def sum3(a, b, c):
            print(f"a={a}")
        print(f"b={b}")
        print(f"c={c}")
        return a+b+c
sum3(1, 2, 3)
```

Arguments can be used in both ways, with or without keyword, if there is no ambiguity

- When used with keyword, they are keyword arguments
- When used without keyword, they are positional arguments

3. What will be the output of the following code snippet?

```
In [ ]: def add(x, y, z=0):
            return x + y + z
print(add(1, 2))
print(add(1, y=2, z=3))
```

4. Why is it beneficial to use docstrings in functions?

- They provide explanations and details about the function for others reading your code. **Both `"""` and `'''` can be used for docstring**

```
In [ ]: # Add docstring and comments to the following function
        def add(x, y, z=0):
    return x + y + z
```

5. How can you see the documentation of a Python library function in the console?

- Use help(library.function)

6. Which of these import statements would avoid a name conflict if there's a local variable `math` in the same script?

- import math as m

7. What will happen if you import the same module multiple times in a Python script?

- Python ignores subsequent imports of the same module in the same script

If you run

```
import myMoudle
```

and then update `myMoudle` and then reload with `import myMoudle` in Jupyter notebook, the module will not be updated. You will need to run

```
from importlib import reload
reload(myModule)
```

or

```
del sys.modules['myMoudle']
```

8. If you want to filter rows in `df` where `age` is greater than 30, which command would you use?

- df[df['age'] > 30]

```
In [ ]: import pandas as pd
        df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie', 'David'],
    'age': [25, 30, 35, 40],
    'height': [165.4, 175.3, 168.5, 180.6]
})
print(df)
df[df['age'] > 30]
```

9. If you want to rename multiple columns in a DataFrame `df`, which method should you use?

- df.rename(columns={'old_col1': 'new_col1', 'old_col2': 'new_col2'})

If you don't specify the key `columns`, it renames the rows

New topic: Regular Expressions

- A regular expression (regex or regexp) is a sequence of characters that defines a search pattern.
- Use case: Regular expressions are used in text processing for searching, matching, and manipulating strings

Examples where regex can play a role

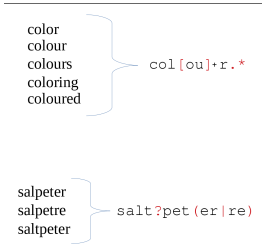
- Find variations in a protein or DNA sequence
    - `"MVR???A"`
    - `"ATG???TAG"`
- American/British spelling, endings and other variants:
    - salpeter, salpetre, saltpeter, nitre, niter or KNO3
    - hemaglobin, heamoglobin, hemaglobins, heamoglobin's
    - catalyze, catalyse, catalyzed...
- A pattern in a VCF file
    - a digit appearing after a tab

Regex is not unique for Python and it is supported by

- most programming languages,
- text editors
- command line tools
- search engines

```
In [ ]: !grep -E "furniture.*sell" ../downloads/blocket_listings.txt
```

Defining a search pattern



- `.` 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

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

`\w+`

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

MORE COMMON OPERATIONS - CLASSES OF CHARACTERS

- `\d` matches any digit

`\d+`

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

MORE COMMON OPERATIONS - CLASSES OF CHARACTERS

- `\s` matches any whitespace (spaces, tabs, ...)

`\s+`

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

More common operations - classes of characters

- `[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 a VCF file

```
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...
```

If we want to find a record with at least one sample (having genotype fields):

`0/0:1:SM` `0/0:4:SM` ...

- `"[01]/[01]"` (or `"\d/\d"`)
- `\s[01]/[01]:`

```
In [ ]: !grep -E "[01]/[01]:" ../downloads/genotypes_small.vcf | head -n 2
```

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 (assuming all homozygous are of the form 1/1, which might not be the case in general)

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

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

```
In [ ]: !grep -E ".*\s1/1:.*\s1/1:.*" ../downloads/genotypes_small.vcf | head -n 1
```

Cheat sheet

- `.` 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

A playground for regex with detailed explanations of your regex

https://regex101.com/

Day 5, Exercise 1 (~30 min)

Practicing regular expressions

- Canvas -> Modules -> Day 5 -> Exercise 1 - day 5

Start the exercise by running

---

python retester.py

---

in the `downloads` folder in a terminal

TAKE A BREAK AFTER THE EXERCISE (~10 MIN)

Session 2

- How to use regex in Python
- Ex2: Regex using Python
- PyQuiz 5.1

Regular expressions in Python

In [ ]:
```
# Import module
import re
```

In [ ]:
```
pattern = "col[ou]+r.*"
text = "The colour of the wall is very vibrant, \
but the color of the sky is even more spectacular."
# Try to find a hit
result = re.search(pattern, text)
print(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 [ ]: print(result.start())
        print(result.end())
print(result.span())
print(f"Found text: '{result.group()}'")
```

```
In [ ]: pattern = "col[ou]+r\w*"
        result = re.search(pattern, text)
print(result)
print(result.span())
print(f"Found text: '{result.group()}'")
```

How to find all occurences of "color" variations?

```
re.finditer
```

In [ ]:
```
pattern = "col[ou]+r\w*"
text = "The colour of the wall is very vibrant, \
but the color of the sky is even more spectacular."

for result in re.finditer(pattern, text):
    print(result)
```

re.compile

In [ ]:
```
# Search pattern in string
pattern = "col[ou]+r\w*"
p_find_colour = re.compile(pattern)
```

In [ ]:
```
p_find_colour.search(text)
```

In [ ]:
```
for result in p_find_colour.finditer(text):
    print(f"Find colour at the position {result.span()}, the word is '{result.group()}'")
```

In [ ]:
```
p_find_colour.findall(text)
```

Benefits of using re.compile

- Improved performance - saves time by compiling the regex once
- Reusability - the compiled regex can be used multiple times
- Early error detection - syntax error of the pattern can be detected at the compiling stage instead of mixed with the other re functions

Case insensitiveness

In [ ]:
```
# Remember, [a-z]+ matches any lower case english word
      p = re.compile('[a-z]+')
result = p.search('123 ATGAAA 456')
print(result)
```

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

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

How to find a full stop?

In [ ]:
```
text = "The first full stop is here: ."
p_find_fullstop = re.compile(".")

result = p_find_fullstop.search(text)
print(f"Found {result.group()} at position {result.start()}")
```

In [ ]:
```
# Use escape character to search
p_find_fullstop = re.compile('\.')

result = p.search(text)
print(f"Found {result.group()} at position {result.start()}")
```

More operations

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

`^hello$`

In [ ]:
```
p_find_hello = re.compile('^hello$')
text = "hello Python"
result = p_find_hello.search(text)
print(result)
```

In [ ]:
```
text = "hello"
result = p_find_hello.search(text)
print(result)
```

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

In [ ]:
```
p_find_salpeter = re.compile('salt?pet(er|re)|nit(er|re)|KNO3')
text = "saltepter or salpeter88 or KNO3 or niter or nitre, \
just too many forms of salpeter!"
for result in p_find_salpeter.finditer(text):
    print(result)
```

Substitution

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

In [ ]:
```
# Spell the word because correctly
       p_fix_because = re.compile('becuase')
p_fix_because.sub('because', text)
print(text)
```

In [ ]:
```
text = p_fix_because.sub('because', text)
       print(text)
```

In [ ]:
```
# Remove additional spaces
       p_remove_extra_space = re.compile('\s+')
p_remove_extra_space.sub(' ', text)
```

Overview

- Construct regular expressions

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

- Searching

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

- Substitution

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

Typical code structure for text matching:

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

Summary

- 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.

Day 5, Exercise 2 (~30 min)

Use regular expressions with Python

- Canvas -> Modules -> Day 5 -> Exercise 2 - day 5

TAKE A BREAK AFTER THE EXERCISE (~10 MIN)

PYQUIZ 5.1 (~10 MIN)

Lunch

Sum up!

Processing files - looping through the lines

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

Store values

```
iterations = 0
information = []

with open('myfile.txt', 'r') as fh:
    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
# variable = literal
```

Compare and membership

```
+, -, *,...   # mathematical
and, or, not  # logical
==, !=        # (in)equality
<, >, <=, >=  # comparison
in            # membership
```

In [ ]:
```
value = 4
        nextvalue = 1
nextvalue += value
print(f"nextvalue: {nextvalue}, value: {value}")
```

In [ ]:
```
x = 5
        y = 7
z = 0

x > 4 or y == 7 and z > 1
```

In [ ]:
```
(x > 4 or y == 7) and z > 1
```

Strings

Works like a list of characters

In [ ]:
```
mystr = "one"
```

In [ ]:
```
mystr += " two" # string concatnation
mystr
```

In [ ]:
```
len(mystr) # get the length
```

In [ ]:
```
"one" in mystr # membership checking
```

String is immutable

In [ ]:
```
mystr = "one"
mystr[1] = "W"
```

In [ ]:
```
mystr = "one"
print(mystr)
mystr = "two"
print(mystr)
```

In [ ]:
```
mystr = "one"
print(f"mystr = {mystr}, address = {id(mystr)}")
mystr = "two"
print(f"mystr = {mystr}, address = {id(mystr)}")
```

String manipulation

```
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...

- Most collections are mutable (not tuple): 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

- List of key value pairs

```
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")
```

In [ ]:
```
set1 = set(["1", "2", "3", "4", "5"])
set1
```

In [ ]:
```
set1.add("1")
set1
```

In [ ]:
```
set2 = set(["3", "6"])
set1.intersection(set2)
```

In [ ]:
```
set1.union(set2)
```

In [ ]:
```
set1.difference(set2)
```

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
```

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) # return as a tuple
length, sequence = find_longest_seq(filepath) # return as two variables
```

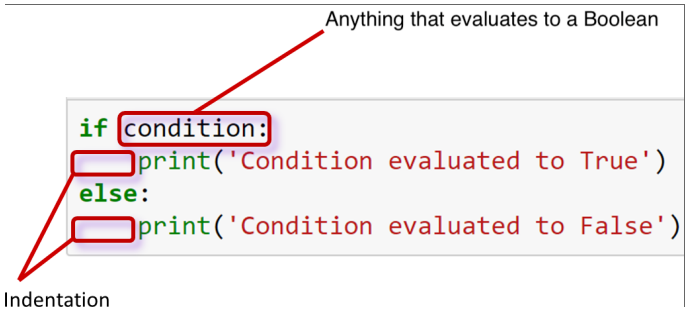
Deciding what to do with if else statement

```
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



`IF X:` IS EQUVALENT TO `IF BOOL(X):`

Program flow - for loops

```
information = []
with open('myfile.txt', 'r') as fh
    for line in fh:
        if is_comment(line):
           use_comment(line)
        else:
           information.append(read_data(line)) # read_data return a list
```

Program flow - while loops

```
information = []
with open('myfile.txt', 'r') as fh:
    # Read the first line
    line = fh.readline()

    # Continue to read lines until an empty string is returned
    while line:
        information.append(read_data(line)) # read_data return a list
        line = file.readline()  # Read the next line
```

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 [ ]:
```
# For loop example
    # You know the number of iterations before hand
user_input = "thank god it's friday"
for letter in user_input:
    print(letter.upper())
```

In [ ]:
```
# While loop example
    # The number of iterations is unknown before hand
i = 0
go_on = True
while go_on:
    c = user_input[i]
    print(c.upper())
    i += 1
    if c == 'd':
        go_on = False
```

Controlling loops

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

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

In [ ]:
```
# example for continue
       user_input = "thank god it's friday"
for letter in user_input:
    if letter == ' ' or letter == '\'':  # Skip spaces and apostrophes
        continue
    print(letter.upper())
```

Watch out!

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

While loops may be infinite!

File Input/Output

- In: Read from files

```
with open(filename, 'r') as fh:
   for line in fh:
       do_stuff(line)
```

  Read information from command line: `sys.argv[1:]`

- Out: Write to files:

```
with open(filename, 'w') as fh:
   fh.write(text)
```

- Printing:

```
print('my_information')
```

Input/Output

Open files should be closed:

```
fh.close()
```

or use the `with` clause

```
with open(filename, "r") as fh:
        do_something
```

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 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, then the arguments are positional
- improve readability when keys are used

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 `mycode.py`
- To run it: `python3 mycode.py` or `python 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

Why programming?

Endless possibilities!

- reverse complement DNA
- custom filtering of VCF files
- plotting of results
- dealing with excel files

Why programming?

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

- Create reproducable results
    - for you and for others to use
- Extract large amount of information

Final advice

- Take a moment to 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
    - try talk about your problem (get a rubber duck **https://en.wikipedia.org/wiki/Rubber_duck_debugging**)
    - search the web
    - Ask AI (such as chatGPT)
    - **NBIS drop-ins**

Now you know Python!

Well done!

JUST A SMALL EXERCISE TO FINISH THE DAY AND HAVE FUN!

Canvas -> Module -> Day 5 -> Exercise 3 - day 5