CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to python week 5

QBio104

Material modified from M. Röttger and A. Schrader

Recap

Local and global variables

Append, sort, delete, split

List comprehensions

Global and Local variables

- Visible in context of variables means, that a variable can be accessed in the current block.
- Variables, that are defined within a function and parameters, are only visible in the **local** area of the function (the function block and all nested block).
- **Global** variables are overlayed by the local definition of a variable with the same identifier. Modifying the local variable is not changing the global variable with the same identifier.

```python
#Here we describe local and global variables

def test_local_variable(DNA_seq_1):
    #remember, variables created or changed inside of a function do not change globally
    DNA_seq_2 = "atgc"
    DNA_seq_1 += DNA_seq_2
    print (f"While running the script:\nDNA_seq_1 > {DNA_seq_1}\nDNA_seq_2 > {DNA_seq_2}\n")
    return DNA_seq_1, DNA_seq_2

DNA_seq_1 = "ATGC"
DNA_seq_2 = ""

print (f"Before running the script:\nDNA_seq_1 > {DNA_seq_1}\nDNA_seq_2 > {DNA_seq_2}\n")

test_local_variable(DNA_seq_1)

print (f"After running the script:\nDNA_seq_1 > {DNA_seq_1}\nDNA_seq_2 > {DNA_seq_2}\n")

#They do change only if you use return statements + assignment
DNA_seq_1, DNA_seq_2 = test_local_variable(DNA_seq_1)

print (f"After running the script and assigning the variables:\nDNA_seq_1 > {DNA_seq_1}\nDNA_seq_2 > {DNA_seq_2}\n")
```

```
Before running the script:
DNA_seq_1 > ATGC
DNA_seq_2 >

While running the script:
DNA_seq_1 > ATGCatgc
DNA_seq_2 > atgc

After running the script:
DNA_seq_1 > ATGC
DNA_seq_2 >

While running the script:
DNA_seq_1 > ATGCatgc
DNA_seq_2 > atgc

After running the script and assigning the variables:
DNA_seq_1 > ATGCatgc
DNA_seq_2 > atgc
```

Append items to a list

- The method `append(...)` can be used to append another object at the end of the list

```
#Here we have an example of how to use the append method
import random

upregulated_genes = []
downregulated_genes = []
gene_expression_foldchange = []

def simulate_expression():
    return random.uniform(-10, 10)

ATP_synthase_genes = ['ATP5A1', 'ATP5B', 'ATP5C1', 'ATP5D', 'ATP5E', \
                      'ATP5F1', 'ATP5G1', 'ATP5G2', 'ATP5G3', 'ATP5H',\
                      'ATP5I', 'ATP5J', 'ATP5J2', 'ATP5L', 'ATP5O']

for gene in range(len(ATP_synthase_genes)):
    gene_expression_foldchange.append(round(simulate_expression(), 2))

print (gene_expression_foldchange)

for gene in range(len(ATP_synthase_genes)):
    if gene_expression_foldchange[gene] > 3:
        upregulated_genes.append(ATP_synthase_genes[gene])
    elif gene_expression_foldchange[gene] < -3:
        downregulated_genes.append(ATP_synthase_genes[gene])

print (f'{len(upregulated_genes)} ATP syntase genes are upregulated')
print (f'{len(downregulated_genes)} ATP syntase genes are downregulated')
print (f'{len(ATP_synthase_genes) - len(upregulated_genes) - len(downregulated_genes)} genes are normally expressed')
```

```
[-8.5, -8.81, 7.51, 5.1, 2.71, -8.2, 0.75, -5.85, -8.43, 1.88, -5.73, -7.93, 1.22, -4.58, -2.36]
2 ATP syntase genes are upregulated
8 ATP syntase genes are downregulated
5 genes are normally expressed
```

Sorting lists

Sorting items within a list

- By using the method `sort(...)`, a list can be sorted.
- The list will be sorted in place, you don't need to assign the object.
- Use the optional argument `reverse=True` to reverse order of sorting.

```
#Here we have an example of how to sort lists

print ("Unsorted list:")
print (gene_expression_foldchange)
gene_expression_foldchange.sort()
print ("Sorted ascending list:")
print (gene_expression_foldchange)
gene_expression_foldchange.sort(reverse=True)
print ("Sorted descending list:")
print (gene_expression_foldchange)
```

```
Unsorted list:
[-7.24, -7.71, 9.72, 4.33, -5.55, 6.7, 5.57, 1.97, 2.65, 3.55, 1.12, -4.14, -2.35, 6.23, -5.45]
Sorted ascending list:
[-7.71, -7.24, -5.55, -5.45, -4.14, -2.35, 1.12, 1.97, 2.65, 3.55, 4.33, 5.57, 6.23, 6.7, 9.72]
Sorted descending list:
[9.72, 6.7, 6.23, 5.57, 4.33, 3.55, 2.65, 1.97, 1.12, -2.35, -4.14, -5.45, -5.55, -7.24, -7.71]
```

List functions

delete items from a list

- To delete an item from a list, the function `del(...)` can be used. The referenced item will be deleted and the list will shrink respectively

```
#Here we have an example of how to use the delete method. be careful when deleting elements as the index "shifts"
import statistics

elephant_population_age = []
for x in range(100):
    elephant_population_age.append(random.randint(0, 70))

print (f'The elephant mean population age is {round(statistics.mean(elephant_population_age), 2)} with {len(elephant_population_age)} individu

elephant_population_age.sort()
while statistics.mean(elephant_population_age) > 25:
    del elephant_population_age[-1]

print (f'The elephant mean population age is {round(statistics.mean(elephant_population_age), 2)} with {len(elephant_population_age)} individu
```

```
The elephant mean population age is 32.42 with 100 individuals
The elephant mean population age is 24.78 with 79 individuals
```

List related methods

Split method turns a string into a list

- The `str.split(sep=someStr)` method takes a string as input and ”splits” it into a list based on a `sep` [separator] argument
- It is often used to split based on metacharacters or other commonly occurring symbols (`.` `,` `:` `/` `|` etc..)

```
#Here we show how the split function works, note that the argument is not present in the output
DNA = "ATCATGCATGCTATATCGTACGGCGCATCAGCACGAGCTAGCAGCGGCTATTCGATCGCGATCGATCGGCGTATCGACAGTCGAGGCA"
digestion_site = "TAT"
DNA_fragments = DNA.split(digestion_site)

print (DNA_fragments)

#You can also use metacharacters as separators
species_string = 'Homo Sapiens\nHomo Erectus\nHomo Neanderthalensis\nHomo Floresiensis\nHomo Naledi'
species_list = species_string.split('\n')
print (species_list)
```

```
['ATCATGCATGC', 'ATCGTACGGCGCATCAGCACGAGCTAGCAGCGGC', 'TCGATCGCGATCGATCGGCG', 'CGACAGTCGAGGCA']
['Homo Sapiens', 'Homo Erectus', 'Homo Neanderthalensis', 'Homo Floresiensis', 'Homo Naledi']
```

List-related methods

The `join` method

- The str method `str.join(someList)` concatenates character string elements in a list separated by a pre-defined `str` delimiter
- All list elements must be str objects

```
#Here we show how the join function works
print (DNA_fragments)
ligation_product = 'N'.join(DNA_fragments)
print (ligation_product)

print ()
print (species_list)
print (f'The species are\n{'\n'.join(species_list)}')
```

```
['ATCATGCATGC', 'ATCGTACGGCGCATCAGCACGAGCTAGCAGCGGC', 'TCGATCGCGATCGATCGGCG', 'CGACAGTCGAGGCA']
ATCATGCATGCNATCGTACGGCGCATCAGCACGAGCTAGCAGCGGCNTCGATCGCGATCGATCGGCGNCGACAGTCGAGGCA

['Homo Sapiens', 'Homo Erectus', 'Homo Neanderthalensis', 'Homo Floresiensis', 'Homo Naledi']
The species are
Homo Sapiens
Homo Erectus
Homo Neanderthalensis
Homo Floresiensis
Homo Naledi
```

List comprehensions

Create and fill a list with one line of code

List comprehension synthax with if/else statements:

`newlist = [expression1 if condition else expression2 for item in iterable]`

List comprehension synthax with **only** if statement:

`newlist = [expression1 for item in iterable if condition]`

```
#Here we explain list comprehension using examples from previous slides

elephant_population_age = []
for x in range(100):
    elephant_population_age.append(random.randint(0, 70))

monkey_population_age = [random.randint(0, 70) for x in range(100)]
```

```
#Here we show how to use list comprehensions using examples from before
#Here is with list comprehension

luminosity = '11010010'
colors = ['green' if x!='0' else 'red' for x in luminosity]
print (colors)

upregulated_genes = [x for x in gene_expression_foldchange if x >3]
downregulated_genes = [x for x in gene_expression_foldchange if x <-3]

print (upregulated_genes)
print (downregulated_genes)
```

```
['green', 'green', 'red', 'green', 'red', 'red', 'green', 'red']
[9.72, 6.7, 6.23, 5.57, 4.33, 3.55]
[-4.14, -5.45, -5.55, -7.24, -7.71]
```

```
#Here we show how to use list comprehensions using examples from before
#here is the previous version

luminosity = '11010010'
colors = map(lambda x: "green" if x !="0" else "red", luminosity)
print (list(colors))

upregulated_genes = []
downregulated_genes = []
for gene in range(len(ATP_synthase_genes)):
    if gene_expression_foldchange[gene] > 3:
        upregulated_genes.append(ATP_synthase_genes[gene])
    elif gene_expression_foldchange[gene] < -3:
        downregulated_genes.append(ATP_synthase_genes[gene])

print (upregulated_genes)
print (downregulated_genes)
```

```
['green', 'green', 'red', 'green', 'red', 'red', 'green', 'red']
['ATP5A1', 'ATP5B', 'ATP5C1', 'ATP5D', 'ATP5E', 'ATP5F1']
['ATP5I', 'ATP5J', 'ATP5J2', 'ATP5L', 'ATP5O']
```

Outlook - Topics

Week 5

- External file handling
- Dictionaries
- Sets
- Set operations

Opening and reading files

Opening a file for reading or writing

- It is possible to interact with python with other files present on the machine. We can open, edit and write files. It all starts with the function `open(file="filename")`
- `open(…)` returns a file object. Files that are opened should be closed once we are done using `fileHandle.close()`
- Files can be opened with different "modes":
  - `open(file="filename", mode='r')` opens the file in read mode (you can access its content but you cannot edit the original file). Opening a file path that does not exist results in error
  - `open(file="filename", mode='w')` opens the file in write mode. This creates a new file or **overwrites** an existing one
- The file object has a `fileHandle.read()` method which casts the file object into a string

Opening and reading files

- It is good habit to specify the full path to the file, including the extension
- If you don't use the full path, the starting point is the directory where you are running the python script from (or jupyter notebook in our case)
- Remember that the open() function returns a file object
- After reading the file, we close the file object, not the string variable!

```
#Here we learn to open files
fasta_file = open('/Users/vittoriotracanna/work/QBIO104/QBio104_VT/Week5/Lecture/wu-hu-1.fasta', 'r')
print ("This is the file object")
print (fasta_file)
fasta_file_string = fasta_file.read()
print ("\nAfter using the .read() method, we have the file as a string data type object")
print (fasta_file_string[:200])
fasta_file.close()
fasta_header_txt = fasta_file_string.split('\n')[0]

#if the file doesn't exist, using the read mode 'r' it will create it
fasta_header_outfile = open('/Users/vittoriotracanna/work/QBIO104/QBio104_VT/Week5/Lecture/header_wu-hu-1.fasta', 'w')
fasta_header_outfile.write(fasta_header_txt)
fasta_header_outfile.close()

#the 'r' mode is the default mode, we can use the .read() already when we open the file
fasta_header_txt = open('/Users/vittoriotracanna/work/QBIO104/QBio104_VT/Week5/Lecture/header_wu-hu-1.fasta').read()
print ('Here is just the header from the files we \n' + fasta_header_txt)
```

```
This is the file object
<_io.TextIOWrapper name='/Users/vittoriotracanna/work/QBIO104/QBio104_VT/Week5/Lecture/wu-hu-1.fasta' mode='r' encoding='UTF-8'>

After using the .read() method, we have the file as a string data type object
>MN908947.3 Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAA
CGAACTTTAAAATCTGTGTGGCTGTCACTCGGC

>MN908947.3 Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome
```

Opening files with `with`

An elegant way to open the file, no need to close it

Synthax for opening a file using the `with` statement:

```
with open('full_file_path.extension') as infile:
    BLOCK
```

You can iterate line by line using the following synthax:

```
with open('full_file_path.extension') as infile:
    for infile_line in infile:
        BLOCK
```

- Note that when using this synthax, in each iteration the loop variable refers to the next line in the file. Each line in the file can only be read one-by-one until all lines have been read once.

Opening files with `readlines()`

List representation of all lines at once

- The function `readlines()` can be used to read all lines at once. It returns a list object containing the lines as elements. Another way is to cast the file handle object into a list.
- Remember, that line ending characters (metacharacter `\n`)are still included in the lines.

```
#here we use with and readlines to open the file

with open('/Users/vittoriotracanna/work/QBIO104/QBio104_VT/Week5/Lecture/short-wu-hu-1.fasta') as short_fasta:
    for fasta_line in short_fasta:
        print (fasta_line)

with open('/Users/vittoriotracanna/work/QBIO104/QBio104_VT/Week5/Lecture/short-wu-hu-1.fasta') as short_fasta:
    fasta_lines = short_fasta.readlines()
print (fasta_lines)
```

```
>MN908947.3 Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome

ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAA

CGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAAC

TAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTG

['>MN908947.3 Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome\n', 'ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCA
ACTTTCGATCTCTTGTAGATCTGTTCTCTAAA\n', 'CGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAAC\n', 'TAATTACTGTCGTTGACAGGACACGAG
TAACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTG\n']
```

CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to Python

Break – 10 minutes

Dictionaries

A structured non-sorted list

- You can think of dictionaries as lists with an alternative index system
- Dictionaries have **keys** (the index identifier) and **values** (what we want to store).
- **Keys** must be **unique** and can be any instance of immutable data type (str, tuple)
- Values can be any python object
- In contrast to other sequence data types, dictionaries are **not sorted**.

Synthax for dictionaries:

dictionary_name = {KEY1: VALUE1, KEY2:VALUE2}

```
#Here we will show basic dictionary construction rules

any_dictionary = {'Key1' : 'values can be strings',
                  'Key2': ['or lists']}
print (any_dictionary)
```

```
{'Key1': 'values can be strings', 'Key2': ['or lists']}
```

Dictionaries

Advanced features

- Since keys can be any immutable data type, we can use tuples
- To refer to an individual value, we can use the key to access it
- We can add key-value pairs to existing dictionaries using the assignment

```
#Here we show more examples of dictionaries

german_cities = {
    "Baden-Württemberg": [("Stuttgart", 634830)],
    "Bayern": [("München", 1488202)],
    "Berlin": [("Berlin", 3669491)],
    "Bremen": [("Bremen", 567559)],
    "Hamburg": [("Hamburg", 1847253)],
    "Hessen": [("Frankfurt am Main", 764104)],
    "Niedersachsen": [("Hannover", 538068)],
    "Nordrhein-Westfalen": [("Köln", 1085664),("Düsseldorf", 635704),
                            ("Dortmund", 588250),("Essen", 582760)],
    }

german_cities["Sachsen"] = [("Leipzig", 609869),("Dresden", 556780)]

print (german_cities["Nordrhein-Westfalen"])
print (german_cities["Sachsen"][1])
```

```
[('Köln', 1085664), ('Düsseldorf', 635704), ('Dortmund', 588250), ('Essen', 582760)]
('Dresden', 556780)
```

Hot question

Recognize data types

```
#Here we show more examples of dictionaries

german_cities = {
    "Baden-Württemberg": [("Stuttgart", 634830)],
    "Bayern": [("München", 1488202)],
    "Berlin": [("Berlin", 3669491)],
    "Bremen": [("Bremen", 567559)],
    "Hamburg": [("Hamburg", 1847253)],
    "Hessen": [("Frankfurt am Main", 764104)],
    "Niedersachsen": [("Hannover", 538068)],
    "Nordrhein-Westfalen": [("Köln", 1085664),("Düsseldorf", 635704),
                            ("Dortmund", 588250),("Essen", 582760)],
    }

german_cities["Sachsen"] = [("Leipzig", 609869),("Dresden", 556780)]

print (german_cities["Nordrhein-Westfalen"])
print (german_cities["Sachsen"][1])
```

```
[('Köln', 1085664), ('Düsseldorf', 635704), ('Dortmund', 588250), ('Essen', 582760)]
('Dresden', 556780)
```

Dictionary methods

`dict.keys()`, `dict.values()` and `dict.items()`

- We can access all the keys of a dictionary as a dict_keys sequence object
- We can access all the values of a dictionary as a dict_values sequence object
- We can access keys and values as list of tuples using the `.items()` method

```
#dictionary values and keys

restriction_enzymes = {
    "EcoRI": "GAATTC","HindIII": "AAGCTT",
    "BamHI": "GGATCC","NotI": "GCGGCCGC",
    "XhoI": "CTCGAG"}

print (restriction_enzymes.keys())
print (restriction_enzymes.values())
enzymes = list(restriction_enzymes.keys())
print (enzymes)
print (restriction_enzymes.items())
```

```
dict_keys(['EcoRI', 'HindIII', 'BamHI', 'NotI', 'XhoI'])
dict_values(['GAATTC', 'AAGCTT', 'GGATCC', 'GCGGCCGC', 'CTCGAG'])
['EcoRI', 'HindIII', 'BamHI', 'NotI', 'XhoI']
dict_items([('EcoRI', 'GAATTC'), ('HindIII', 'AAGCTT'), ('BamHI', 'GGATCC'), ('NotI', 'GCGGCCGC'), ('XhoI', 'CTCGAG')])
```

Iterate over dictionaries

We can iterate over the keys or both keys and values

- We can iterate over the keys of a dictionary. This returns a key iterable
- We can iterate over both keys and values using the method .items() but we need to declare two variables in the for loop

```
#Iterating over dictionaries

for enzyme_name in restriction_enzymes.keys():
    print (f"{enzyme_name} recognizes the DNA sequence '{restriction_enzymes[enzyme_name]}'")

for enzyme_name, enzyme_DNA_target in restriction_enzymes.items():
    print (f"{enzyme_DNA_target} is the target DNA sequence for '{enzyme_name}'")
```

```
EcoRI recognizes the DNA sequence 'GAATTC'
HindIII recognizes the DNA sequence 'AAGCTT'
BamHI recognizes the DNA sequence 'GGATCC'
NotI recognizes the DNA sequence 'GCGGCCGC'
XhoI recognizes the DNA sequence 'CTCGAG'
GAATTC is the target DNA sequence for 'EcoRI'
AAGCTT is the target DNA sequence for 'HindIII'
GGATCC is the target DNA sequence for 'BamHI'
GCGGCCGC is the target DNA sequence for 'NotI'
CTCGAG is the target DNA sequence for 'XhoI'
```

Dictionary comprehension

Just like for lists, we can create one-line dictionaries

- We use the same structure we use for lists but replace [] with {}
- Can also use conditional statements, just like list comprehensions

```
#Dictionary comprehensions

german_cities = {
    "Berlin": 3669491, "Hamburg": 1847253, "München": 1488202,
    "Köln": 1085664, "Frankfurt am Main": 764104,"Stuttgart": 634830,
    "Düsseldorf": 635704, "Leipzig": 609869, "Dortmund": 588250,
    "Essen": 582760, "Bremen": 567559, "Dresden": 556780,
    "Hannover": 538068}

large_cities = {city: population for city, population in german_cities.items() if population > 1000000}
small_cities = {city: population for city, population in german_cities.items() if city not in large_cities}

print (f'{', '.join(large_cities.keys())} are large cities')
print (f'{', '.join(small_cities.keys())} are small cities')
```

```
Berlin, Hamburg, München, Köln are large cities
Frankfurt am Main, Stuttgart, Düsseldorf, Leipzig, Dortmund, Essen, Bremen, Dresden, Hannover are small cities
```

Sets

The last piece in python data types

- A set object consists of an unordered collection of **immutable** and **unique** elements.
- A set can be defined by embedding the elements within curly brackets {…}
- A set can contain mixed object types (as long as they are immutable, no lists!)

```
#Here are the basics of sets

nucleotide_set = {'A', 'T', 'C', 'G'}
print (nucleotide_set)

nucleotide_redundant_list = ['A','A','T','T','C','C','G','G']
print (set(nucleotide_redundant_list))

import random
random_numbers_list = [random.randrange(0, 10, 2) for x in range(1000000)]
print (len(random_numbers_list), set(random_numbers_list))

print (set([0, 'mix', ('of', 'immutable', 'objects')]))
```

```
{'T', 'A', 'C', 'G'}
{'T', 'A', 'C', 'G'}
1000000 {0, 2, 4, 6, 8}
{0, 'mix', ('of', 'immutable', 'objects')}
```

Adding sets

There are multiple ways to create and add sets

- We can make a set from a string, each character becomes an element
- We can use the `set.add(…)` method or the operand `|` (pipe character)
- `set.add(…)` works inplace (no need for assignment) `|` instead needs assignment

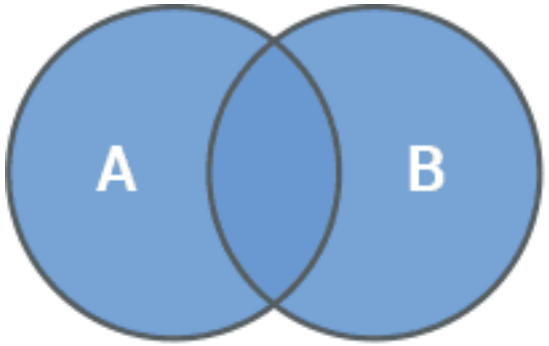
```
#Here we test some set functions to add sets

nucleotides = set('AT')
print (nucleotides)

nucleotides.add('G')
print (nucleotides)

nucleotides = nucleotides | {'C'} # |= also works
print (nucleotides)
```

```
{'T', 'A'}
{'T', 'A', 'G'}
{'T', 'A', 'G', 'C'}
```

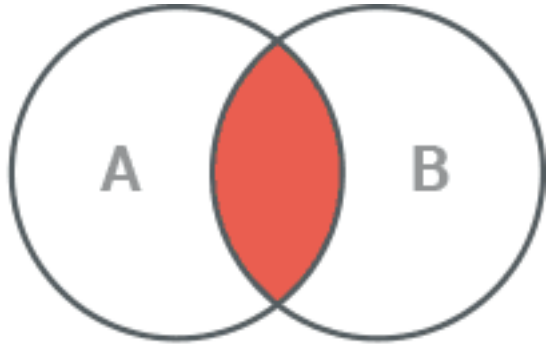
Set theory (Mengenlehre)

Long time no see

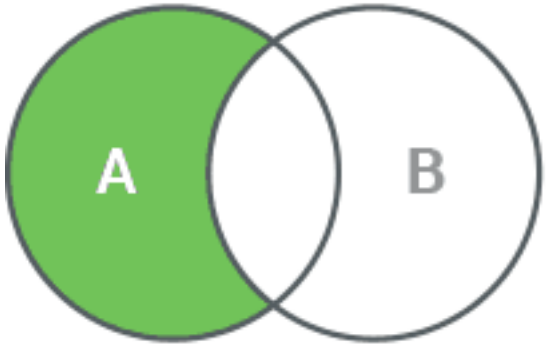
- For union between sets
  - `A | B`
  - `A.union(B)`
- For intersection between sets
  - `A & B`
  - `A.intersection(B)`
- For difference between sets
  - `A - B`
  - `A.difference(B)`
- Symmetric difference (disjunction)
  - `A ^ B`
  - `A.symmetric_difference(B)`



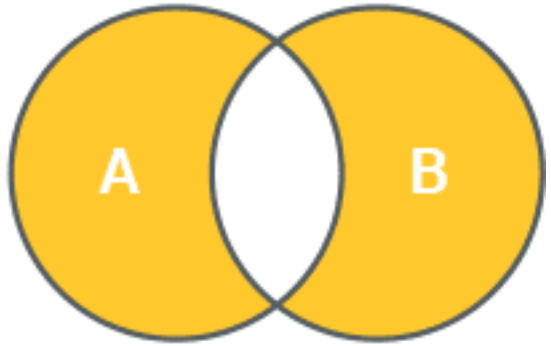
Union



Intersection



Difference

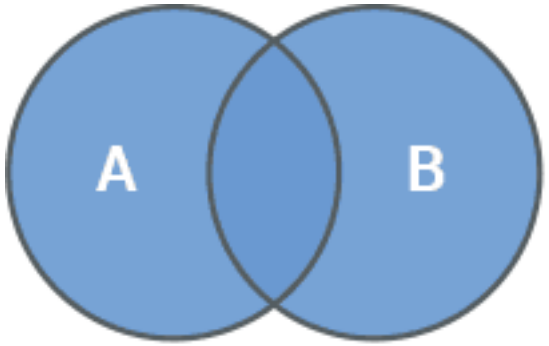


Symmetric Difference

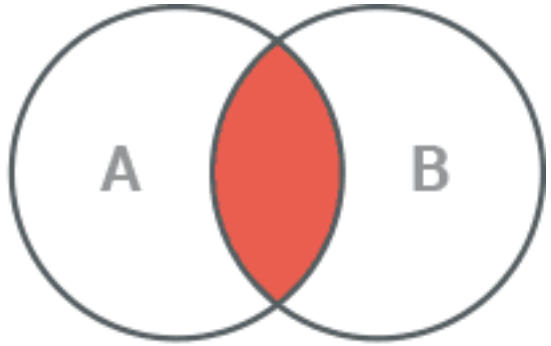
Set theory examples

```python
# set operations in python

range_5 = set(range(6))
even_6 = set(range(0, 7, 2))
print (range_5)
print (even_6)

#union
print (f'Union: {range_5 | even_6}')

#intersection
print (f'Intersection: {range_5 & even_6}')

#difference
print (f'Difference: {range_5 - even_6}')

#symmetric difference
print (f'Symmetric difference: {range_5 ^ even_6}')
```

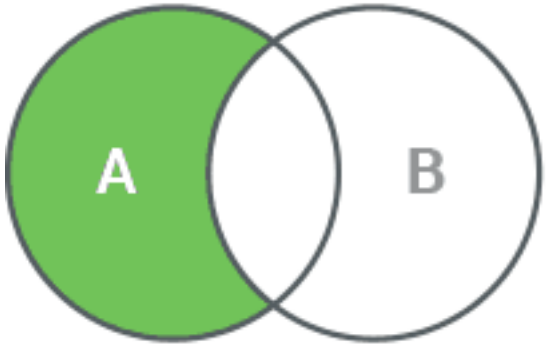
```
{0, 1, 2, 3, 4, 5}
{0, 2, 4, 6}
```



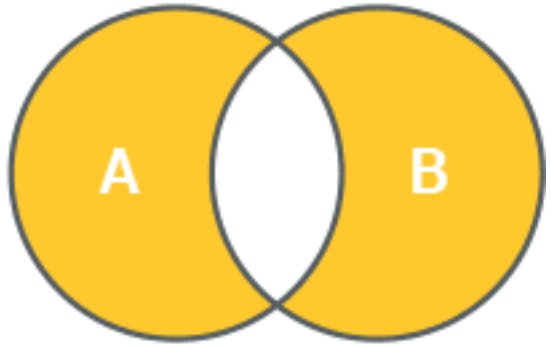
Union



Intersection



Difference

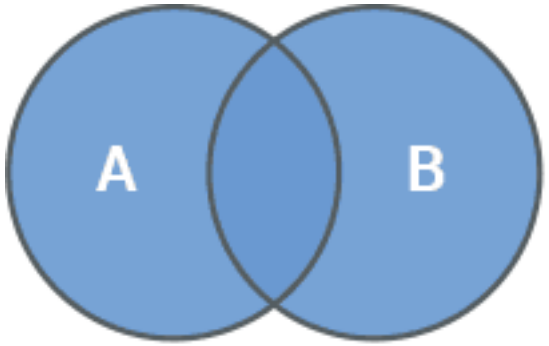


Symmetric Difference

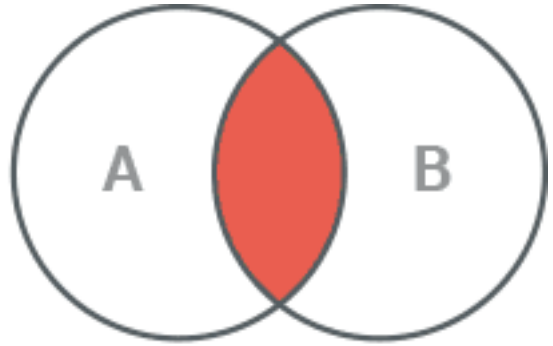
Set theory examples

```python
# set operations in python

range_5 = set(range(6))
even_6 = set(range(0, 7, 2))
print (range_5)
print (even_6)

#union
print (f'Union: {range_5 | even_6}')

#intersection
print (f'Intersection: {range_5 & even_6}')

#difference
print (f'Difference: {range_5 - even_6}')

#symmetric difference
print (f'Symmetric difference: {range_5 ^ even_6}')
```

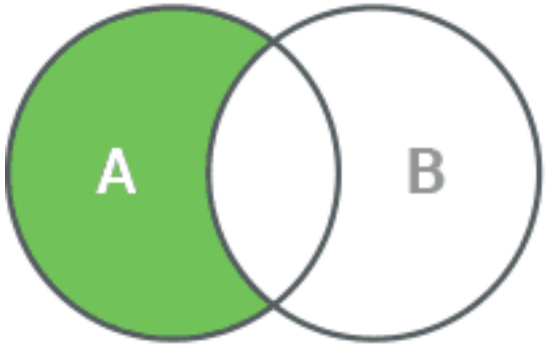
```
{0, 1, 2, 3, 4, 5}
{0, 2, 4, 6}
Union: {0, 1, 2, 3, 4, 5, 6}
Intersection: {0, 2, 4}
Difference: {1, 3, 5}
Symmetric difference: {1, 3, 5, 6}
```



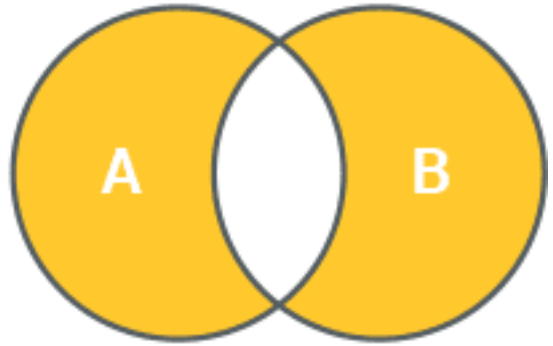
Union



Intersection



Difference



Symmetric Difference

CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to Python

Break – 10 minutes

How do you decide what data type to use

From each according to his ability, to each according to his needs

- Numerics:
  - int Integer number: `1103`
  - float Floating point number: `3.141592653589793`
  - bool Boolean type, `True` or `False`
- Sequences:
  - str Character string: `"Hello world"`
  - list List: `[1, 2, 3, 4]`
  - tuple Tuple: `('a', 'b', 'c')`
- Mappings:
  - set Set: `{'A', 'T', 'C', 'G'}`
  - dict Dictionary: `{'Kölsch': 2, 'Altbier': 1}`

Examples using different data types

Let's build together a script that compares DNA sequences from a fasta file

- Each sequence entry in a fasta file (.fasta, .fa) is characterised by two sections. A header (starts with ">") and a sequence (can be nucleotides or amino acids)
- We want to make a script that:
  - opens the fasta file
  - Stores the fasta information in a structured way
  - Iterates over each sequence and compares to every other sequence
  - We will take percentage identity as a characteristic (% of the sequence that is identical)
  - Stores the information in a structured way
  - Save the information by writing it into a file

Compare DNA sequences

Open the fasta file

- For simplicity, we will say that the `unknown_sequences.fna` file is in the working directory
- What mode do we need to use with the `open()` function?
- What can we print to make sure it works fine?
- What data type do we expect?

Compare DNA sequences

Open the fasta file

- For simplicity, we will say that the `unknown_sequences.fna` file is in the working directory
- What mode do we need to use with the `open()` function?
- What can we print to make sure it works fine?
- What data type do we expect?

```
#Week5 lecture example for using data types

#Open and read the file in read mode
unknown_sequences = open('unknown_sequences.fna','r').read()
print (unknown_sequences)
print (type(unknown_sequences))
```

```
>Unknown_sequence_1
TCAAAGCCGCACATTGTCCCGCTATTGCTATTACATATTATACCCAGTAA
>Unknown_sequence_2
CTCCATAGTATACCGATTCATCGGCGCCTTGCATCATCAAGGGTTGTATA
>Unknown_sequence_3
CGGTTAAGTCAACCCTCCCGCTCACACTCCGTACCTCGGTACTCTACAGG
>Unknown_sequence_4
CTCCATTGTATACTGTTGCATTGGGCCTTTGCATCGTTAAGGTCCGGATA
>Unknown_sequence_5
GATGGGTGTATATGGTTGATGCGATTGGCAATACAGATCAAGCCGTCCCT
>Unknown_sequence_6
CCCCGTTGTATCCCGTCCCATCGTGCCCGTGCAGCATTAAAGGTCATATA
>Unknown_sequence_7
GGATATCAGTTTGGGATTCCGGGCGTGCCAACTAAAAGTGTCCCTTTGAG
>Unknown_sequence_8
GTACAGAATTTCGGCTCACCGCGCATGCGGCTACTCATATATCCGCAAAT
>Unknown_sequence_9
ATCTACCGGATAGGATCGTGACGAGTGCCACTACCGACCGCTTCTTCTAT
>Unknown_sequence_10
CTCCGTTGTAGGCCGTCGCATCGGGCCCGTGCAACATTGAGCGTCGCATC

<class 'str'>
```

Compare DNA sequences

Stores the fasta information in a structured way

- What data type fits our needs?
- How would you go about organizing it?
- What can we print to check if everything works?

Compare DNA sequences

Stores the fasta information in a structured way

- What data type fits our needs?
- How would you go about organizing it?
- What can we print to check if everything works?

```
#Week5 lecture example for using data types
#Note that re-running this cell before running the previous one will result in an error
#Do you know why?

#Stores the fasta information in a structured way

unknown_sequences = unknown_sequences.split('\n')
print (unknown_sequences)

sequence_dictionary = {}
#Why do we use len(unknown_sequences-1)? look at the previous print statement output
for line in range(0, len(unknown_sequences)-1, 2):
    header = unknown_sequences[line]
    sequence = unknown_sequences[line+1]
    sequence_dictionary[header] = sequence

print (sequence_dictionary)
```

```
['>Unknown_sequence_1', 'TCAAAGCCGCACATTGTCCCGCTATTGCTATTACATATTATACCCAGTAA', '>Unknown_s
equence_2', 'CTCCATAGTATACCGATTCATCGGCGCCTTGCATCATCAAGGGTTGTATA', '>Unknown_sequence_3',
'CGGTTAAGTCAACCCTCCCGCTCACACTCCGTACCTCGGTACTCTACAGG', '>Unknown_sequence_4', 'CTCCATTGTAT
ACTGTTGCATTGGGCCTTTGCATCGTTAAGGTCCGGATA', '>Unknown_sequence_5', 'GATGGGTGTATATGGTTGATGCG
ATTGGCAATACAGATCAAGCCGTCCCT', '>Unknown_sequence_6', 'CCCCGTTGTATCCCGTCCCATCGTGCCCGTGCAGC
ATTAAAGGTCATATA', '>Unknown_sequence_7', 'GGATATCAGTTTGGGATTCCGGGCGTGCCAACTAAAAGTGTCCCTTT
GAG', '>Unknown_sequence_8', 'GTACAGAATTTCGGCTCACCGCGCATGCGGCTACTCATATATCCGCAAAT', '>Unkn
own_sequence_9', 'ATCTACCGGATAGGATCGTGACGAGTGCCACTACCGACCGCTTCTTCTAT', '>Unknown_sequence
_10', 'CTCCGTTGTAGGCCGTCGCATCGGGCCCGTGCAACATTGAGCGTCGCATC', '']
{'>Unknown_sequence_1': 'TCAAAGCCGCACATTGTCCCGCTATTGCTATTACATATTATACCCAGTAA', '>Unknown_s
equence_2': 'CTCCATAGTATACCGATTCATCGGCGCCTTGCATCATCAAGGGTTGTATA', '>Unknown_sequence_3':
'CGGTTAAGTCAACCCTCCCGCTCACACTCCGTACCTCGGTACTCTACAGG', '>Unknown_sequence_4': 'CTCCATTGTAT
ACTGTTGCATTGGGCCTTTGCATCGTTAAGGTCCGGATA', '>Unknown_sequence_5': 'GATGGGTGTATATGGTTGATGCG
ATTGGCAATACAGATCAAGCCGTCCCT', '>Unknown_sequence_6': 'CCCCGTTGTATCCCGTCCCATCGTGCCCGTGCAGC
ATTAAAGGTCATATA', '>Unknown_sequence_7': 'GGATATCAGTTTGGGATTCCGGGCGTGCCAACTAAAAGTGTCCCTTT
GAG', '>Unknown_sequence_8': 'GTACAGAATTTCGGCTCACCGCGCATGCGGCTACTCATATATCCGCAAAT', '>Unkn
own_sequence_9': 'ATCTACCGGATAGGATCGTGACGAGTGCCACTACCGACCGCTTCTTCTAT', '>Unknown_sequence
_10': 'CTCCGTTGTAGGCCGTCGCATCGGGCCCGTGCAACATTGAGCGTCGCATC'}
```

Compare DNA sequences

Iterate over each sequence and compare

- Iterates over each sequence and compares to every other sequence
- We will take percentage identity as a characteristic (% of the sequence that is identical)

Compare DNA sequences

Iterate over each sequence and compare

- Iterates over each sequence and compares to every other sequence
- We will take percentage identity as a characteristic (% of the sequence that is identical)

```
#Week5 lecture example for using data types
#Iterates over each sequence and compares to every other sequence
#We will take percentage identity as a characteristic (% of the sequence that is identical)

#make a function to calculate sequence identity
def percentage_identity(seq1, seq2):
    #all sequences have the same length
    identical_pos = 0
    for x in range(len(seq1)):
        if seq1[x] == seq2[x]:
            identical_pos+=1
    return identical_pos/len(seq1)

#iterate over all the sequences
for header1 in sequence_dictionary.keys():
    sequence1 = sequence_dictionary[header1]
    #since we are comparing all vs all, we iterate again
    for header2 in sequence_dictionary.keys():
        print (f'analysing {header1} and {header2}:')
        sequence2 = sequence_dictionary[header2]
        identity = percentage_identity(sequence1, sequence2)
```

```
analysing >Unknown_sequence_1 and >Unknown_sequence_1:
analysing >Unknown_sequence_1 and >Unknown_sequence_2:
analysing >Unknown_sequence_1 and >Unknown_sequence_3:
analysing >Unknown_sequence_1 and >Unknown_sequence_4:
analysing >Unknown_sequence_1 and >Unknown_sequence_5:
analysing >Unknown_sequence_1 and >Unknown_sequence_6:
analysing >Unknown_sequence_1 and >Unknown_sequence_7:
analysing >Unknown_sequence_1 and >Unknown_sequence_8:
analysing >Unknown_sequence_1 and >Unknown_sequence_9:
analysing >Unknown_sequence_1 and >Unknown_sequence_10:
analysing >Unknown_sequence_2 and >Unknown_sequence_1:
analysing >Unknown_sequence_2 and >Unknown_sequence_2:
analysing >Unknown_sequence_2 and >Unknown_sequence_3:
analysing >Unknown_sequence_2 and >Unknown_sequence_4:
```

Compare DNA sequences

Store the % identity information in a structured format and write to file

- Stores the information in a structured way
- Write it into a file

Compare DNA sequences

Store the % identity information in a structured format and write to file

- Stores the information in a structured way
- Write it into a file

```
#Week5 lecture example for using data types
#Iterates over each sequence and compares to every other sequence
#We will take percentage identity as a characteristic (% of the sequence that is identical)
#Store it in a dictionary

pairwise_id_dict = {}

for header1 in sequence_dictionary.keys():
    sequence1 = sequence_dictionary[header1]
    #since we are comparing all vs all, we iterate again
    for header2 in sequence_dictionary.keys():
        sequence2 = sequence_dictionary[header2]
        identity = percentage_identity(sequence1, sequence2)
        pairwise_id_dict[(header1, header2)] = identity

#structure it for output
output_text = ''
for sequences, perc_id in pairwise_id_dict.items():
    output_text += f'{sequences[0]}\t{sequences[1]}\t{perc_id}\n'
print (output_text)

#write to file
output_file = open('unknown_sequences_perc_id.tsv', 'w')
output_file.write(output_text)
output_file.close()
```

```
>Unknown_sequence_1     >Unknown_sequence_1     1.0
>Unknown_sequence_1     >Unknown_sequence_2     0.18
>Unknown_sequence_1     >Unknown_sequence_3     0.22
>Unknown_sequence_1     >Unknown_sequence_4     0.24
>Unknown_sequence_1     >Unknown_sequence_5     0.36
>Unknown_sequence_1     >Unknown_sequence_6     0.24
>Unknown_sequence_1     >Unknown_sequence_7     0.38
>Unknown_sequence_1     >Unknown_sequence_8     0.38
>Unknown_sequence_1     >Unknown_sequence_9     0.32
>Unknown_sequence_1     >Unknown_sequence_10    0.14
>Unknown_sequence_2     >Unknown_sequence_1     0.18
>Unknown_sequence_2     >Unknown_sequence_2     1.0
>Unknown_sequence_2     >Unknown_sequence_3     0.3
>Unknown_sequence_2     >Unknown_sequence_4     0.72
```

Compare DNA sequences

Put it all together

```python
#Week5 lecture example for using data types
import sys

def fasta_to_dict(splitted_fasta):
    #parses a line-split fasta file into a dictionary
    sequence_dictionary = {}
    for line in range(0, len(splitted_fasta)-1, 2):
        header = splitted_fasta[line]
        sequence = splitted_fasta[line+1]
        sequence_dictionary[header] = sequence
    return sequence_dictionary

def percentage_identity(seq1, seq2):
    #this function calculates sequence identity
    #all sequences have the same length
    identical_pos = 0
    for x in range(len(seq1)):
        if seq1[x] == seq2[x]:
            identical_pos+=1
    return identical_pos/len(seq1)

def structure_output(pairwise_id_dict):
    #structure the dictionary for output as tsv
    output_text = ''
    for sequences, perc_id in pairwise_id_dict.items():
        output_text += f'{sequences[0]}\t{sequences[1]}\t{perc_id}\n'
    return output_text

if __name__ == '__main__':
    if len(sys.argv)>1:
        #this let's us parse any fasta file that the user provides
        #if adding a fasta file path when running the script
        unknown_sequences = open(sys.argv[1], 'r').read()
    else:
        #Open and read the file in read mode, default behaviour
        unknown_sequences = open('unknown_sequences.fna','r').read()

    #Stores the fasta information in a structured way
    unknown_sequences = unknown_sequences.split('\n')

    #parse the fasta to a dictionary
    sequence_dictionary = fasta_to_dict(unknown_sequences)

    pairwise_id_dict = {}
    #iterate over all the sequences
    for header1 in sequence_dictionary.keys():
        sequence1 = sequence_dictionary[header1]
        #since we are comparing all vs all, we iterate again
        for header2 in sequence_dictionary.keys():
            sequence2 = sequence_dictionary[header2]
            identity = percentage_identity(sequence1, sequence2)
            pairwise_id_dict[(header1, header2)] = identity

    output_text = structure_output(pairwise_id_dict)
    print (output_text)

    #write to file
    output_file = open('unknown_sequences_perc_id.tsv', 'w')
    output_file.write(output_text)
    output_file.close()
```

```
(base) vittoriotracanna@MacBook-Pro-von-Vittorio QBio104_VT % cd Week5/Lecture
(base) vittoriotracanna@MacBook-Pro-von-Vittorio Lecture % python3 parse_perc_id.py
>Unknown_sequence_1	>Unknown_sequence_1	1.0
>Unknown_sequence_1	>Unknown_sequence_2	0.18
>Unknown_sequence_1	>Unknown_sequence_3	0.22
>Unknown_sequence_1	>Unknown_sequence_4	0.24
>Unknown_sequence_1	>Unknown_sequence_5	0.36
>Unknown_sequence_1	>Unknown_sequence_6	0.24
>Unknown_sequence_1	>Unknown_sequence_7	0.38
>Unknown_sequence_1	>Unknown_sequence_8	0.38
>Unknown_sequence_1	>Unknown_sequence_9	0.32
>Unknown_sequence_1	>Unknown_sequence_10	0.14
>Unknown_sequence_2	>Unknown_sequence_1	0.18
>Unknown_sequence_2	>Unknown_sequence_2	1.0
>Unknown_sequence_2	>Unknown_sequence_3	0.3
>Unknown_sequence_2	>Unknown_sequence_4	0.72
>Unknown_sequence_2	>Unknown_sequence_5	0.24
>Unknown_sequence_2	>Unknown_sequence_6	0.68
>Unknown_sequence_2	>Unknown_sequence_7	0.28
>Unknown_sequence_2	>Unknown_sequence_8	0.26
>Unknown_sequence_2	>Unknown_sequence_9	0.28
>Unknown_sequence_2	>Unknown_sequence_10	0.66
>Unknown_sequence_3	>Unknown_sequence_1	0.22
>Unknown_sequence_3	>Unknown_sequence_2	0.3
>Unknown_sequence_3	>Unknown_sequence_3	1.0
```

Recap

Open files

Write files

New data type: Dictionary

dict.keys(), dict.values(), dict.items()

New data type: Sets

set.union(), set.intersection(), set.difference(), set.symmetric_difference()

Live exercise using different data types