CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to python week 4

QBio104

Material modified from M. Röttger and A. Schrader

Recap week 3

Logical connectives
for-loops
while-loops

Logical connectives

`and`, `or`, `not`, `xor`

- Logical connectives `and`, `or`, `not` and `xor` (ExCLUSIVE `or`) can be used to logically connect Boolean input values with a Boolean output value.
- Using logical connectives, very complex logical expressions can be build up.

`and` needs both terms being compared to be `True`

`or` need either or both the terms being compared to be `True`

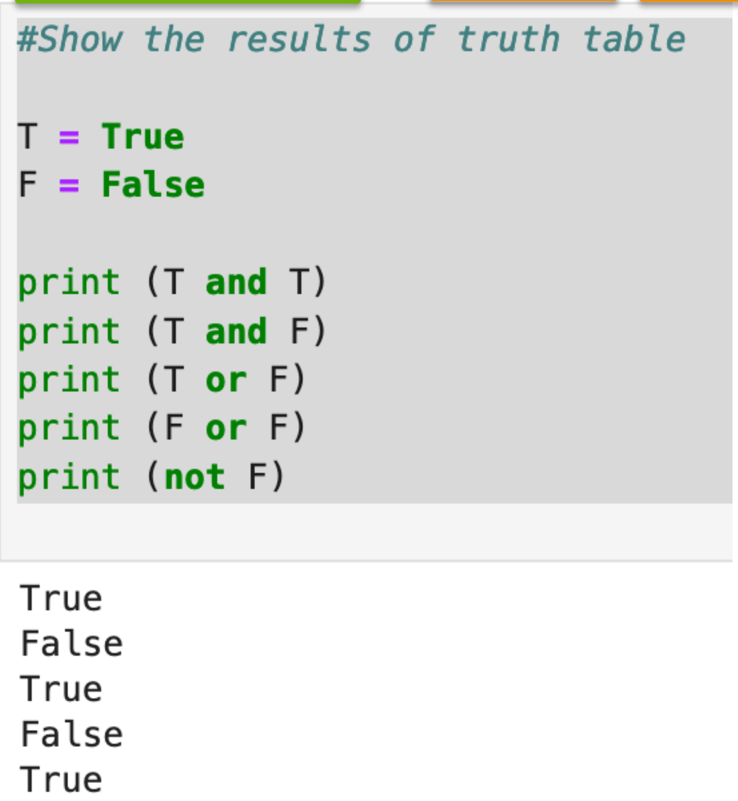
`not` refers to one term and is `True` only when the term is `False` (0 or an empty `str` is considered `False` in Boolean operators)

`xor` need either (not both!) the terms being compared to be `True`

Logical connector truth table

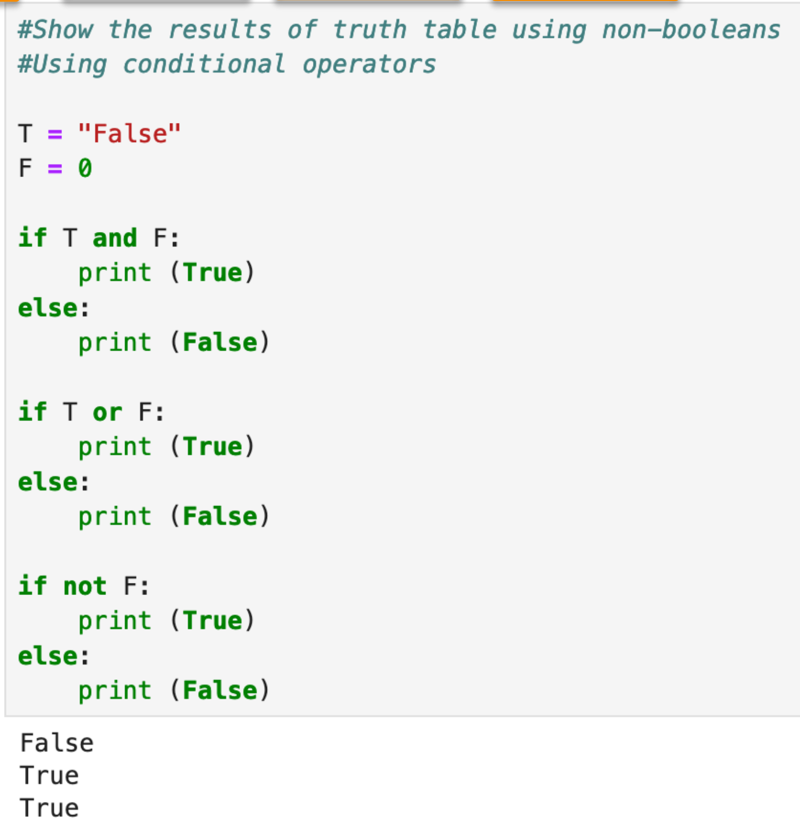
Given A and B, what is the result of the statement

A	B	A == B	A != B	A and B	A or B *	not A
False	False	True	False	False	False	True
False	True	False	True	False	True	True
True	False	False	True	False	True	False
True	True	True	False	True	True	False



```
#Show the results of truth table

T = True
F = False

print (T and T)
print (T and F)
print (T or F)
print (F or F)
print (not F)
```

```
True
False
True
False
True
```



```
#Show the results of truth table using non-booleans
#Using conditional operators

T = "False"
F = 0

if T and F:
    print (True)
else:
    print (False)

if T or F:
    print (True)
else:
    print (False)

if not F:
    print (True)
else:
    print (False)
```

```
False
True
True
```

`for`-loop

How it works

- The loop iterates over the single elements in a sequence.
- In each iteration, the loop variable references the next object in the sequence type object.
- The function `range(…)` can be used to generate a sequence of integer numbers that can be used as sequence object in a loop.
- Beyond the examples given here, we will always use the `range(…)` function. As it provides a consistent sequence that iterates over a variable with a known type (integer)

`while`-loop

Condition-based loop

- Before each iteration of a `while`-loop, a condition is evaluated
- If the respective condition is `True`, the while-block will be executed.
- Once executed, the condition is evaluated again.. And so on.
- If the initial condition is `False` the loop will not be executed at all

Syntax of the `while`-loop:

```
while CONDITION:
    BLOCK
```

Outlook

Week 4

- Build better functions
- Lambda functions
- Local and global variables
- List methods
- List comprehensions
- Dictionaries
- Tuples

Building better functions

What we know so far

- The Syntax of a function definition:

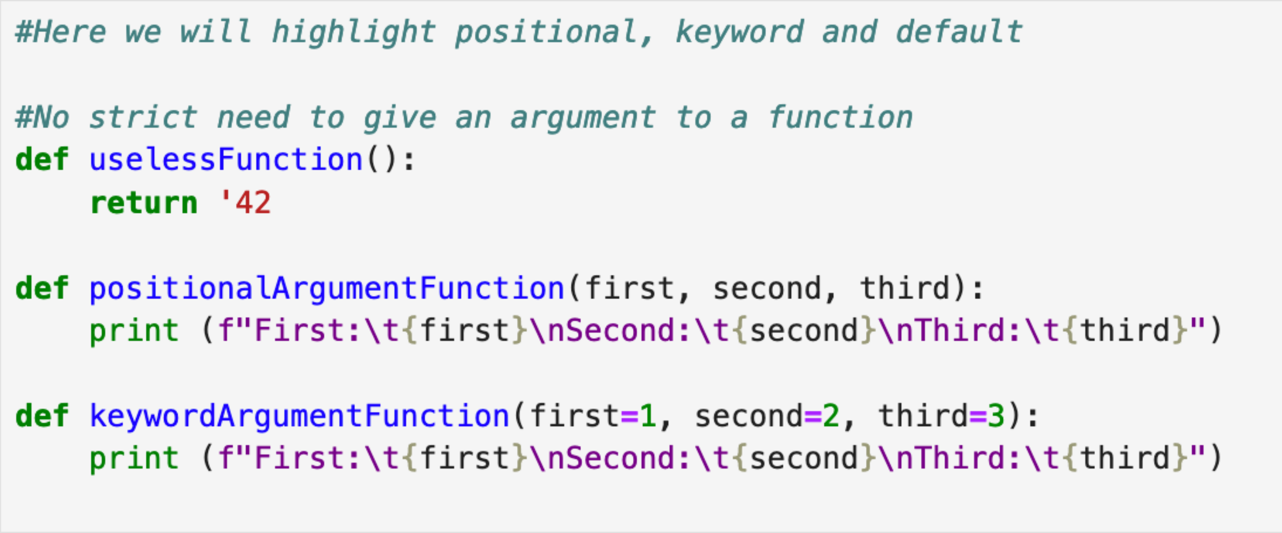
```
def FUNCTIONNAME( PARAMETER1, PARAMETER2, … ):
    ‟FUNCTION DOCUMENTATION‟
    STATEMENTS
return EXPRESSION
```

- The return is not needed, if absent it will return “None”

Function arguments

Positional and keyword arguments

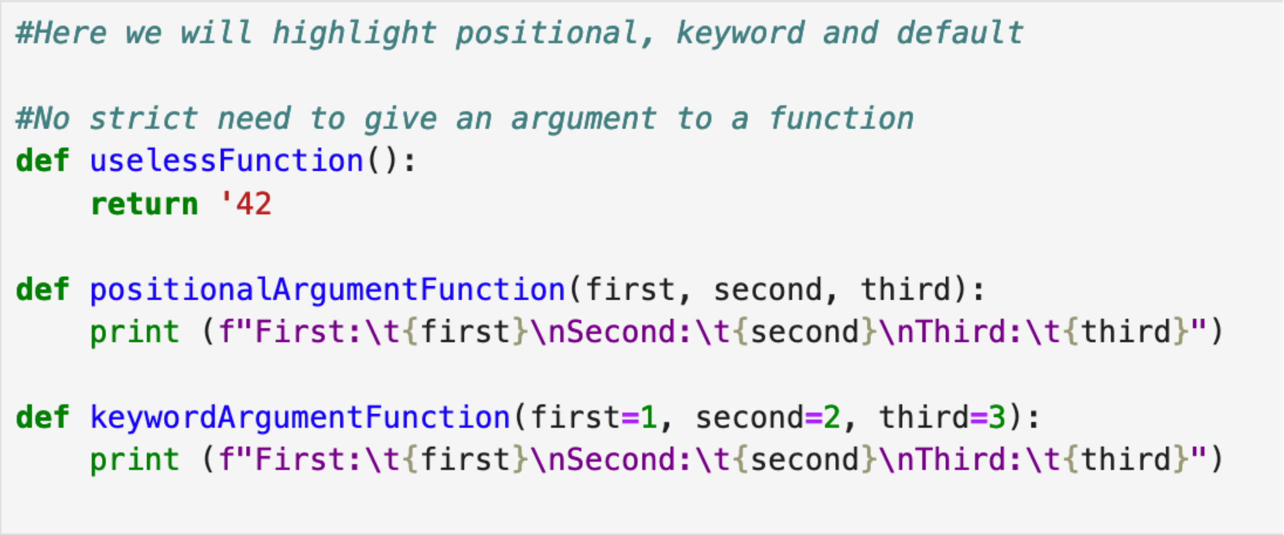
- You can provide default values, the function will use that value if the user does not provide one instead
- Positional arguments are arguments that are read and interpreted by a function based on their order
- Keyword arguments are arguments that have to be provided using the matching keyword, if the keyword is not provided, they behave like positional arguments.



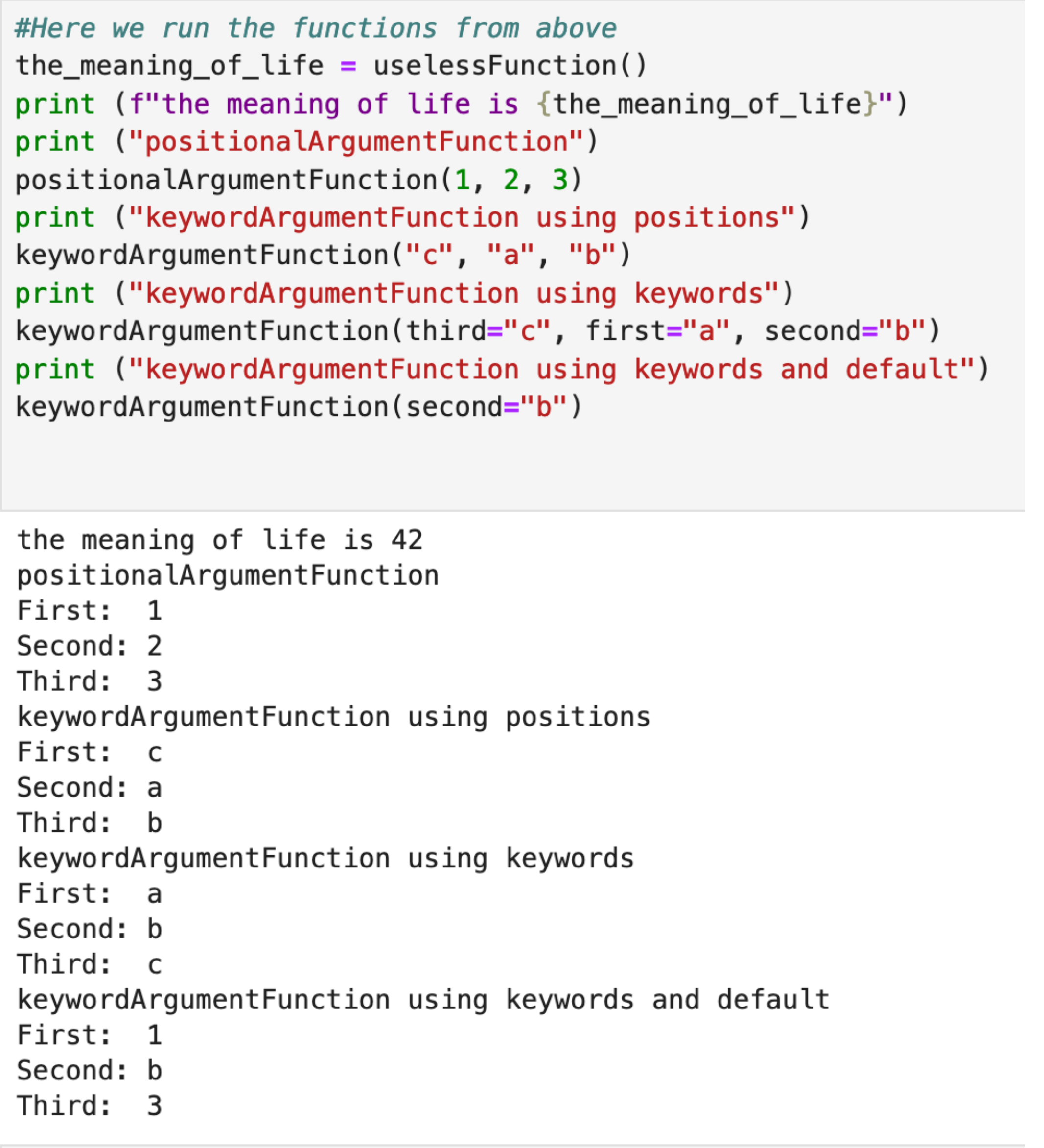
```python
#Here we will highlight positional, keyword and default

#No strict need to give an argument to a function
def uselessFunction():
    return '42

def positionalArgumentFunction(first, second, third):
    print (f"First:\t{first}\nSecond:\t{second}\nThird:\t{third}")

def keywordArgumentFunction(first=1, second=2, third=3):
    print (f"First:\t{first}\nSecond:\t{second}\nThird:\t{third}")
```

Function arguments

Positional and keyword arguments



```
#Here we will highlight positional, keyword and default

#No strict need to give an argument to a function
def uselessFunction():
    return '42

def positionalArgumentFunction(first, second, third):
    print (f"First:\t{first}\nSecond:\t{second}\nThird:\t{third}")

def keywordArgumentFunction(first=1, second=2, third=3):
    print (f"First:\t{first}\nSecond:\t{second}\nThird:\t{third}")
```

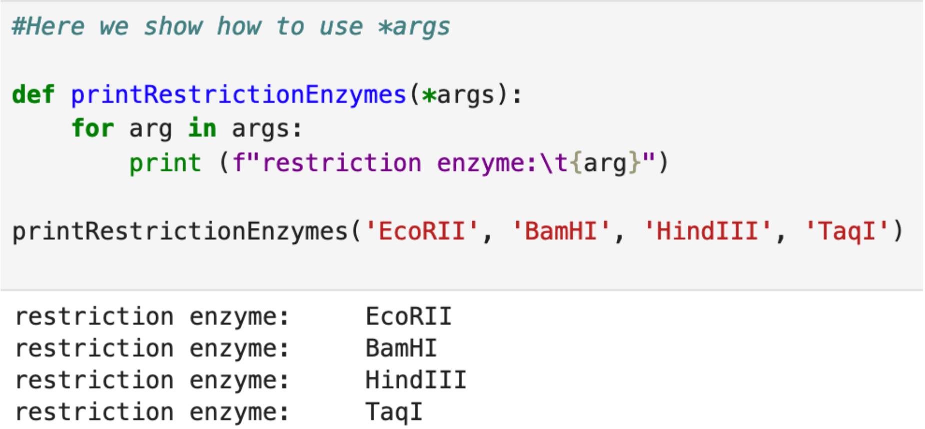


```
#Here we run the functions from above
the_meaning_of_life = uselessFunction()
print (f"the meaning of life is {the_meaning_of_life}")
print ("positionalArgumentFunction")
positionalArgumentFunction(1, 2, 3)
print ("keywordArgumentFunction using positions")
keywordArgumentFunction("c", "a", "b")
print ("keywordArgumentFunction using keywords")
keywordArgumentFunction(third="c", first="a", second="b")
print ("keywordArgumentFunction using keywords and default")
keywordArgumentFunction(second="b")
```

```
the meaning of life is 42
positionalArgumentFunction
First:  1
Second: 2
Third:  3
keywordArgumentFunction using positions
First:  c
Second: a
Third:  b
keywordArgumentFunction using keywords
First:  a
Second: b
Third:  c
keywordArgumentFunction using keywords and default
First:  1
Second: b
Third:  3
```

Wildcard arguments

- *args allow you to pass any number of arguments to a function. You can use it when you do not know how many arguments will be passed to your function.
- You don’t need to use *args as only the * (asterisk) is necessary. Writing *args and is just a convention.
- Arguments inside a function are stored in a list and can be accessed with a for loop.



```python
#Here we show how to use *args

def printRestrictionEnzymes(*args):
    for arg in args:
        print (f"restriction enzyme:\t{arg}")

printRestrictionEnzymes('EcoRII', 'BamHI', 'HindIII', 'TaqI')
```

```
restriction enzyme:	EcoRII
restriction enzyme:	BamHI
restriction enzyme:	HindIII
restriction enzyme:	TaqI
```

Introduction to Python

Break – 10 minutes

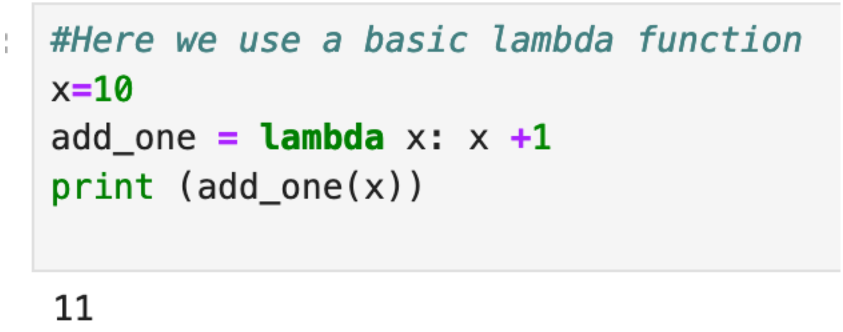
`Lambda` function

Anonymous function

- By using the keyword `lambda`, it is possible to create anonymous functions.
- `lambda` functions can take arguments.
- `lambda` functions return via an expression (no `return` statement).

Synthax of `lambda` function:

`lambda` var : STATEMENT



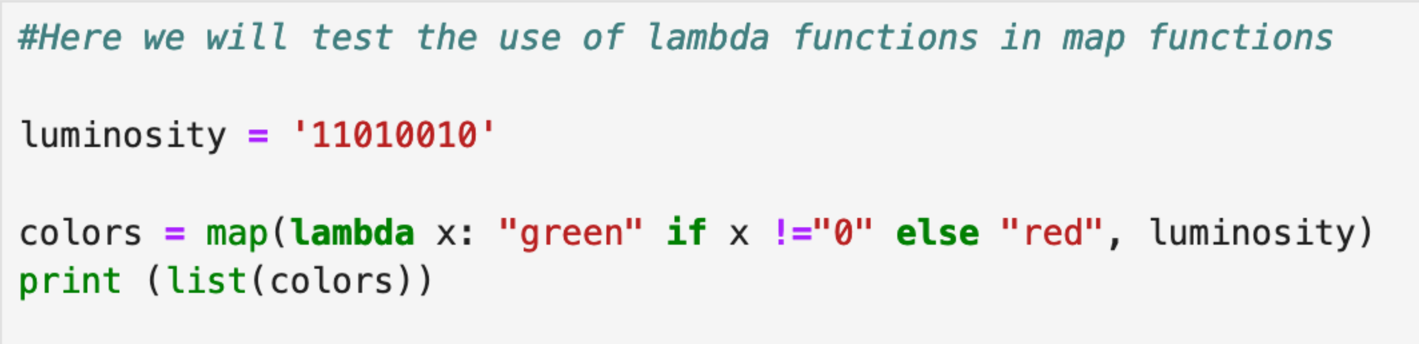
```
#Here we use a basic lambda function
x=10
add_one = lambda x: x +1
print (add_one(x))
```

```
11
```

map function

Some functions allow functions as arguments

- The `map()` function is applying a specific function on a sequence and returns a list containing the respective results as elements
- A limited number of other functions can accept functions as input
- This is an excellent use case for lambda functions that can be defined directly when calling `map()`



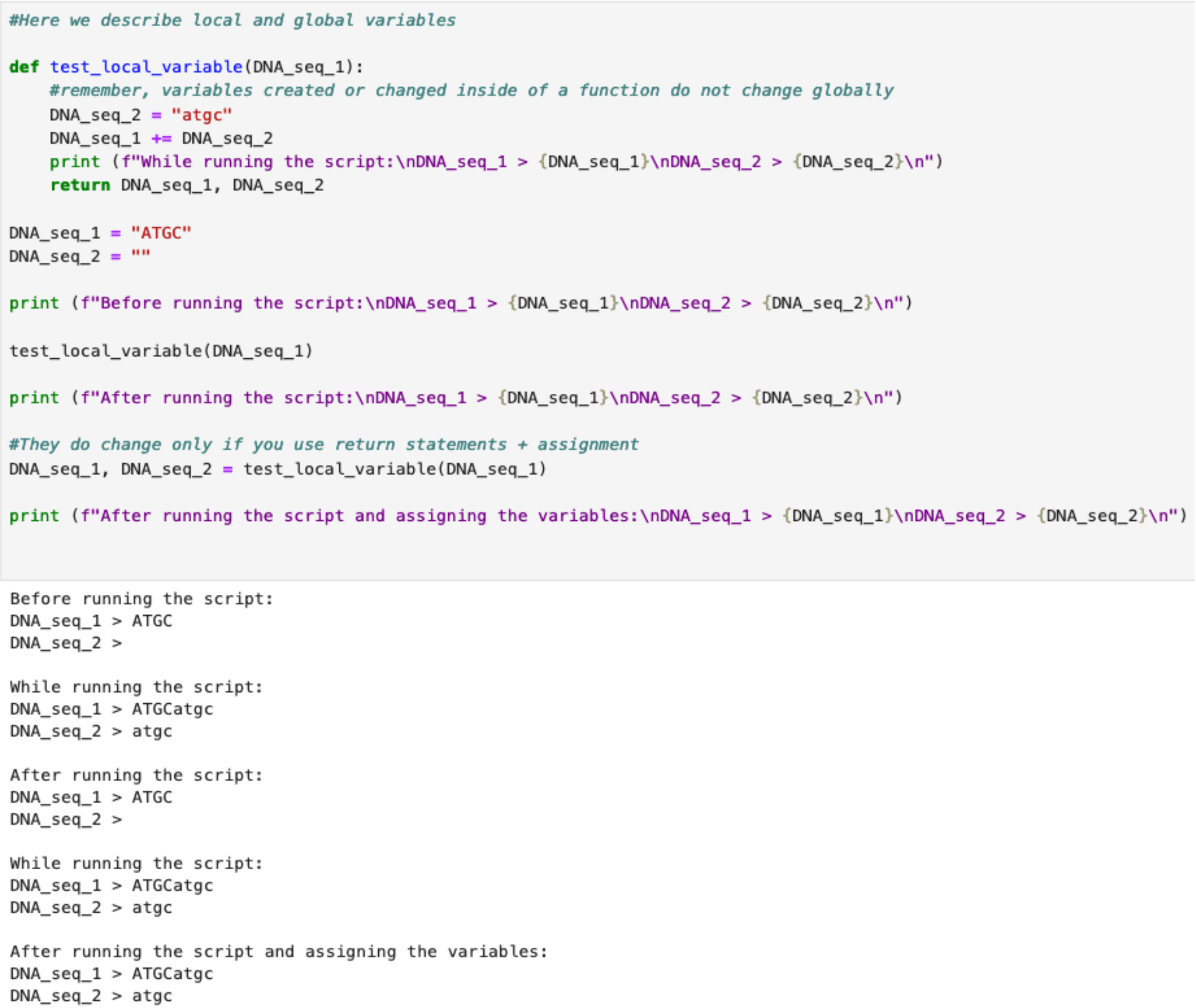
```
#Here we will test the use of lambda functions in map functions

luminosity = '11010010'

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

- `map()` returns an object that can be printed when its type is `list`

Scope of variables

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.

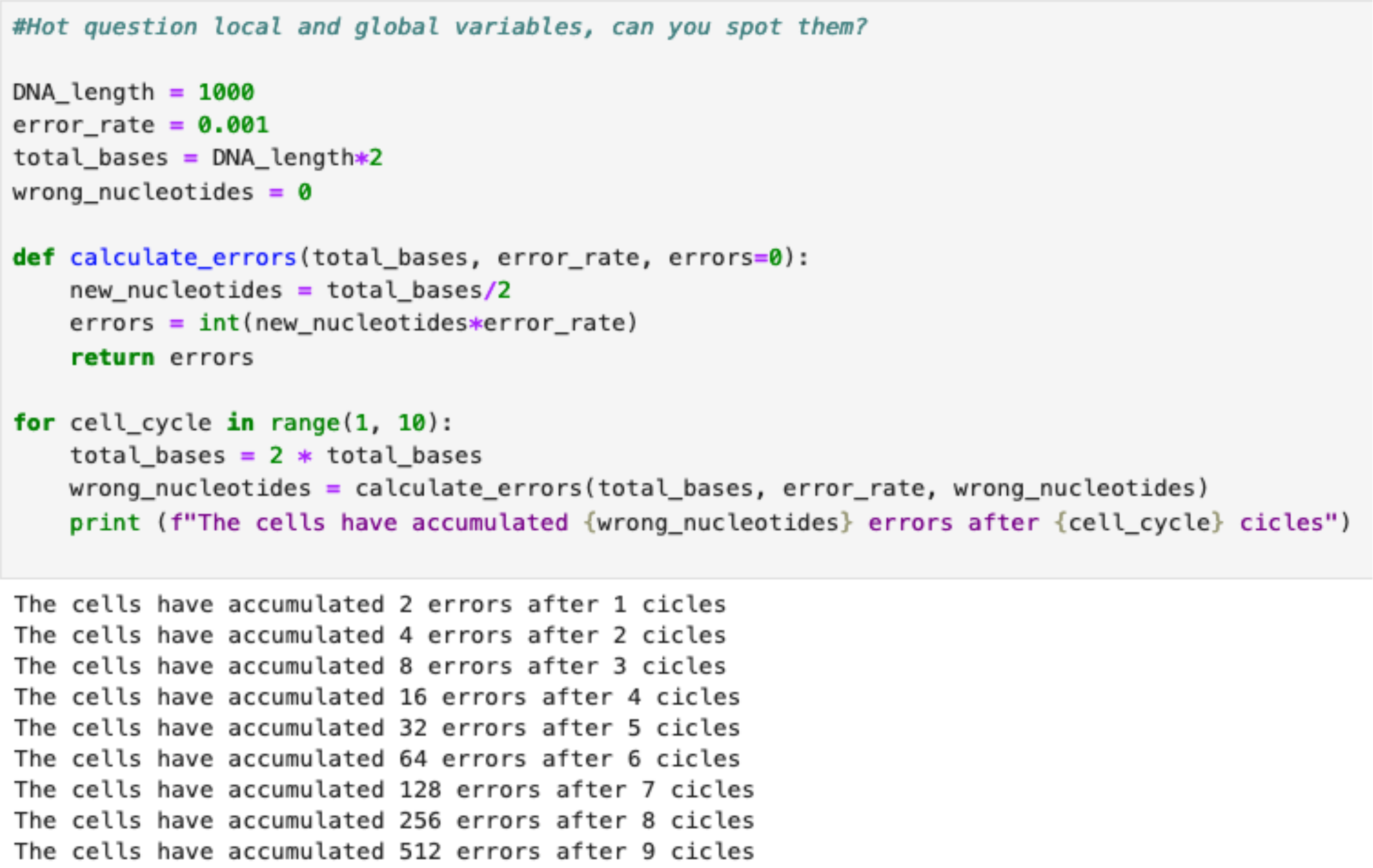


```
#Here we describe local and global variables

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
```

Hot question

Local and global variables, spot the variables

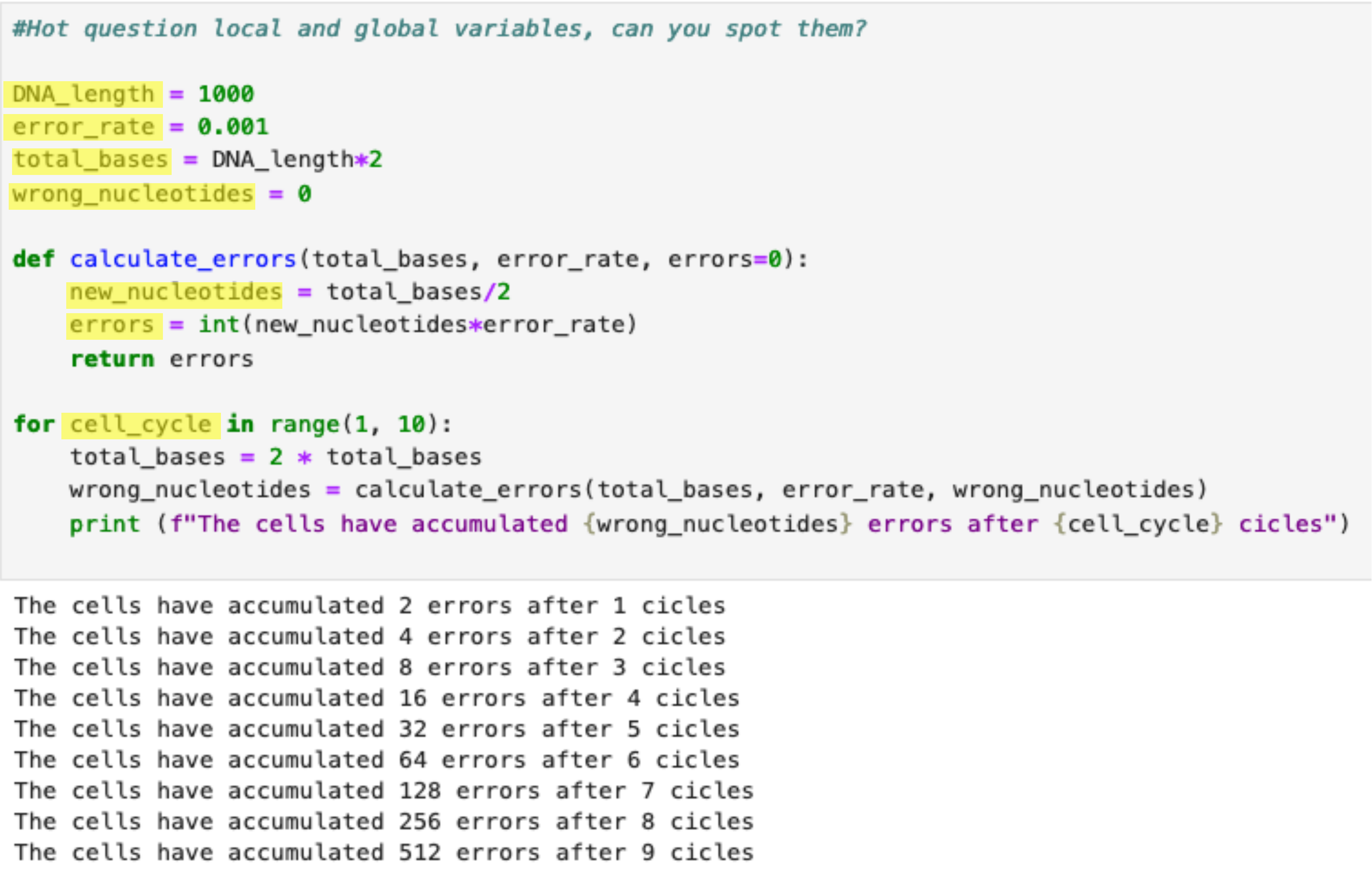


```python
#Hot question local and global variables, can you spot them?

DNA_length = 1000
error_rate = 0.001
total_bases = DNA_length*2
wrong_nucleotides = 0

def calculate_errors(total_bases, error_rate, errors=0):
    new_nucleotides = total_bases/2
    errors = int(new_nucleotides*error_rate)
    return errors

for cell_cycle in range(1, 10):
    total_bases = 2 * total_bases
    wrong_nucleotides = calculate_errors(total_bases, error_rate, wrong_nucleotides)
    print (f"The cells have accumulated {wrong_nucleotides} errors after {cell_cycle} cicles")
```

```
The cells have accumulated 2 errors after 1 cicles
The cells have accumulated 4 errors after 2 cicles
The cells have accumulated 8 errors after 3 cicles
The cells have accumulated 16 errors after 4 cicles
The cells have accumulated 32 errors after 5 cicles
The cells have accumulated 64 errors after 6 cicles
The cells have accumulated 128 errors after 7 cicles
The cells have accumulated 256 errors after 8 cicles
The cells have accumulated 512 errors after 9 cicles
```

Hot question

Local and global variables, global or local?

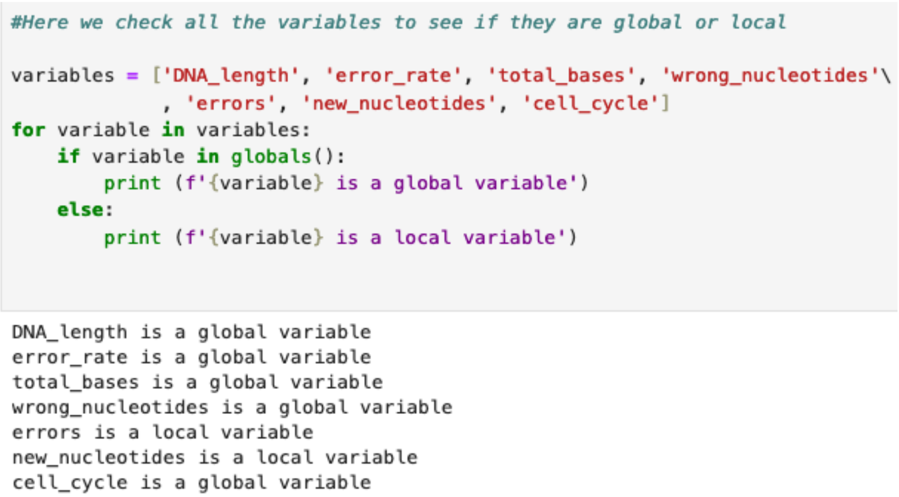


```
#Hot question local and global variables, can you spot them?

DNA_length = 1000
error_rate = 0.001
total_bases = DNA_length*2
wrong_nucleotides = 0

def calculate_errors(total_bases, error_rate, errors=0):
    new_nucleotides = total_bases/2
    errors = int(new_nucleotides*error_rate)
    return errors

for cell_cycle in range(1, 10):
    total_bases = 2 * total_bases
    wrong_nucleotides = calculate_errors(total_bases, error_rate, wrong_nucleotides)
    print (f"The cells have accumulated {wrong_nucleotides} errors after {cell_cycle} cicles")
```

```
The cells have accumulated 2 errors after 1 cicles
The cells have accumulated 4 errors after 2 cicles
The cells have accumulated 8 errors after 3 cicles
The cells have accumulated 16 errors after 4 cicles
The cells have accumulated 32 errors after 5 cicles
The cells have accumulated 64 errors after 6 cicles
The cells have accumulated 128 errors after 7 cicles
The cells have accumulated 256 errors after 8 cicles
The cells have accumulated 512 errors after 9 cicles
```

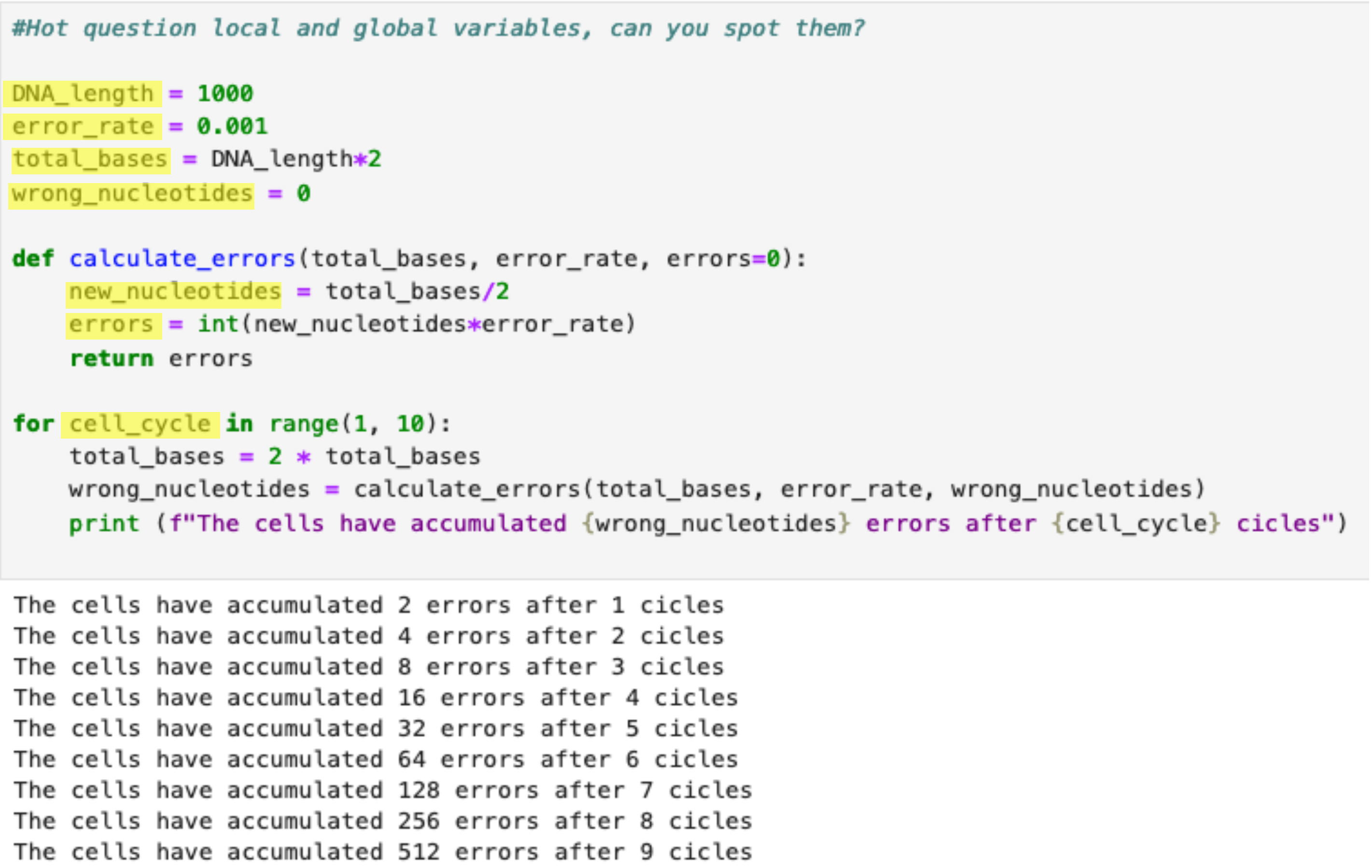
Hot question

Global and local



```
#Here we check all the variables to see if they are global or local

variables = ['DNA_length', 'error_rate', 'total_bases', 'wrong_nucleotides'\
            , 'errors', 'new_nucleotides', 'cell_cycle']
for variable in variables:
    if variable in globals():
        print (f'{variable} is a global variable')
    else:
        print (f'{variable} is a local variable')
```

```
DNA_length is a global variable
error_rate is a global variable
total_bases is a global variable
wrong_nucleotides is a global variable
errors is a local variable
new_nucleotides is a local variable
cell_cycle is a global variable
```



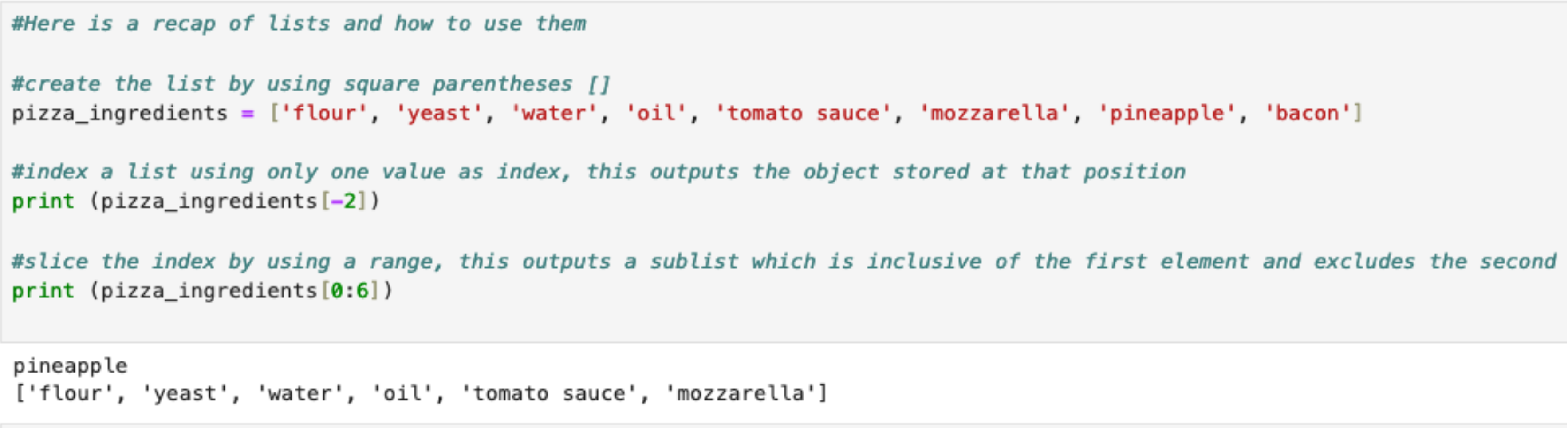
```
#Hot question local and global variables, can you spot them?

DNA_length = 1000
error_rate = 0.001
total_bases = DNA_length*2
wrong_nucleotides = 0

def calculate_errors(total_bases, error_rate, errors=0):
    new_nucleotides = total_bases/2
    errors = int(new_nucleotides*error_rate)
    return errors

for cell_cycle in range(1, 10):
    total_bases = 2 * total_bases
    wrong_nucleotides = calculate_errors(total_bases, error_rate, wrong_nucleotides)
    print (f"The cells have accumulated {wrong_nucleotides} errors after {cell_cycle} cicles")
```

```
The cells have accumulated 2 errors after 1 cicles
The cells have accumulated 4 errors after 2 cicles
The cells have accumulated 8 errors after 3 cicles
The cells have accumulated 16 errors after 4 cicles
The cells have accumulated 32 errors after 5 cicles
The cells have accumulated 64 errors after 6 cicles
The cells have accumulated 128 errors after 7 cicles
The cells have accumulated 256 errors after 8 cicles
The cells have accumulated 512 errors after 9 cicles
```

Delving into lists

Recap

- A list object can hold an **ordered** list of elements. These elements can be any combination of Python objects.
- Lists are defined by the use of square backets
  - EXAMPLE_LIST = [“Object1”, ”Object2”, …]
- Accessing elements in a list can be accomplished by indexing or slicing, whereas slicing created a sub-list of the original list



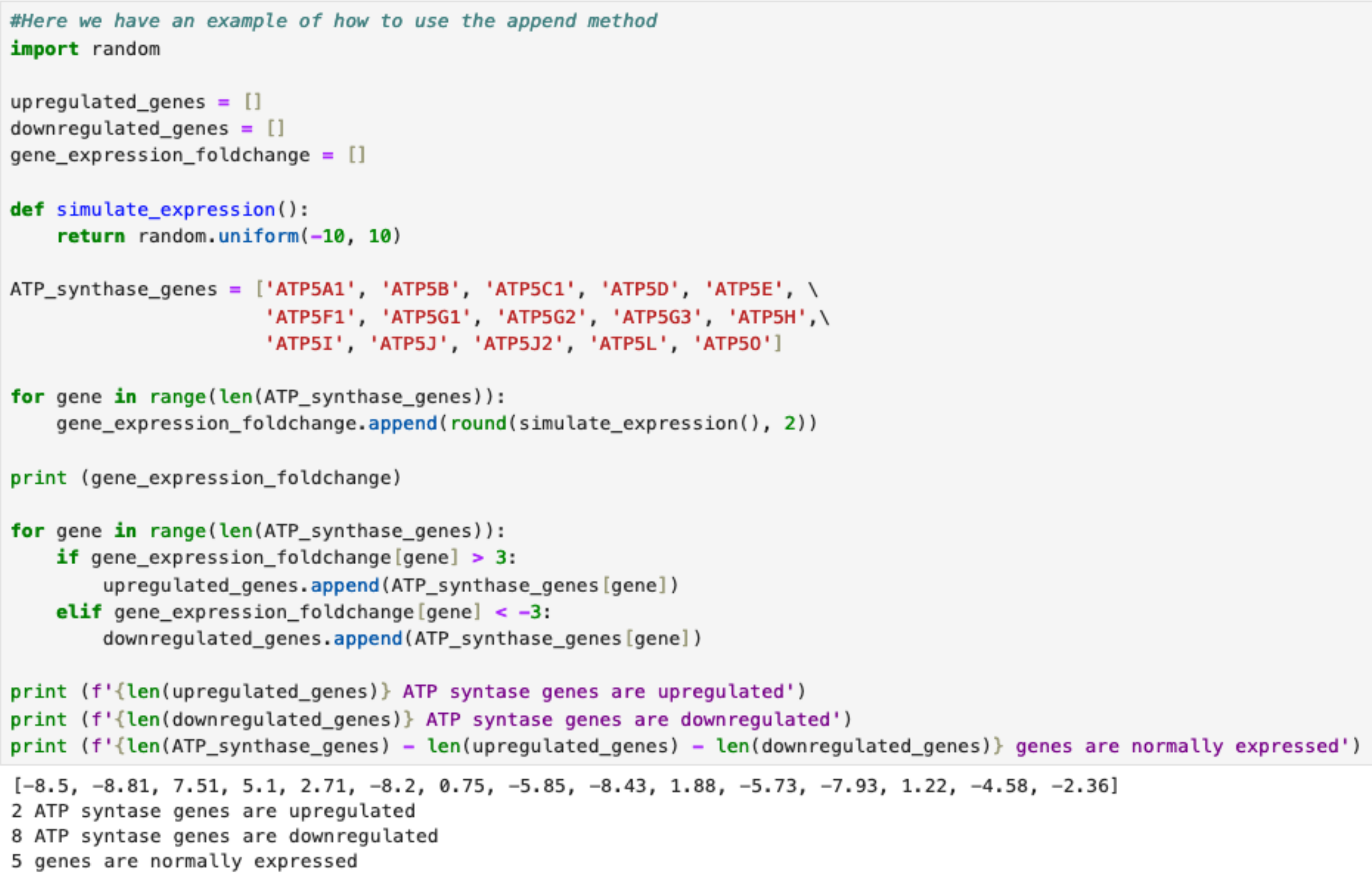
```
#Here is a recap of lists and how to use them

#create the list by using square parentheses []
pizza_ingredients = ['flour', 'yeast', 'water', 'oil', 'tomato sauce', 'mozzarella', 'pineapple', 'bacon']

#index a list using only one value as index, this outputs the object stored at that position
print (pizza_ingredients[-2])

#slice the index by using a range, this outputs a sublist which is inclusive of the first element and excludes the second
print (pizza_ingredients[0:6])
```

```
pineapple
['flour', 'yeast', 'water', 'oil', 'tomato sauce', 'mozzarella']
```

List methods

Append items to a list

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



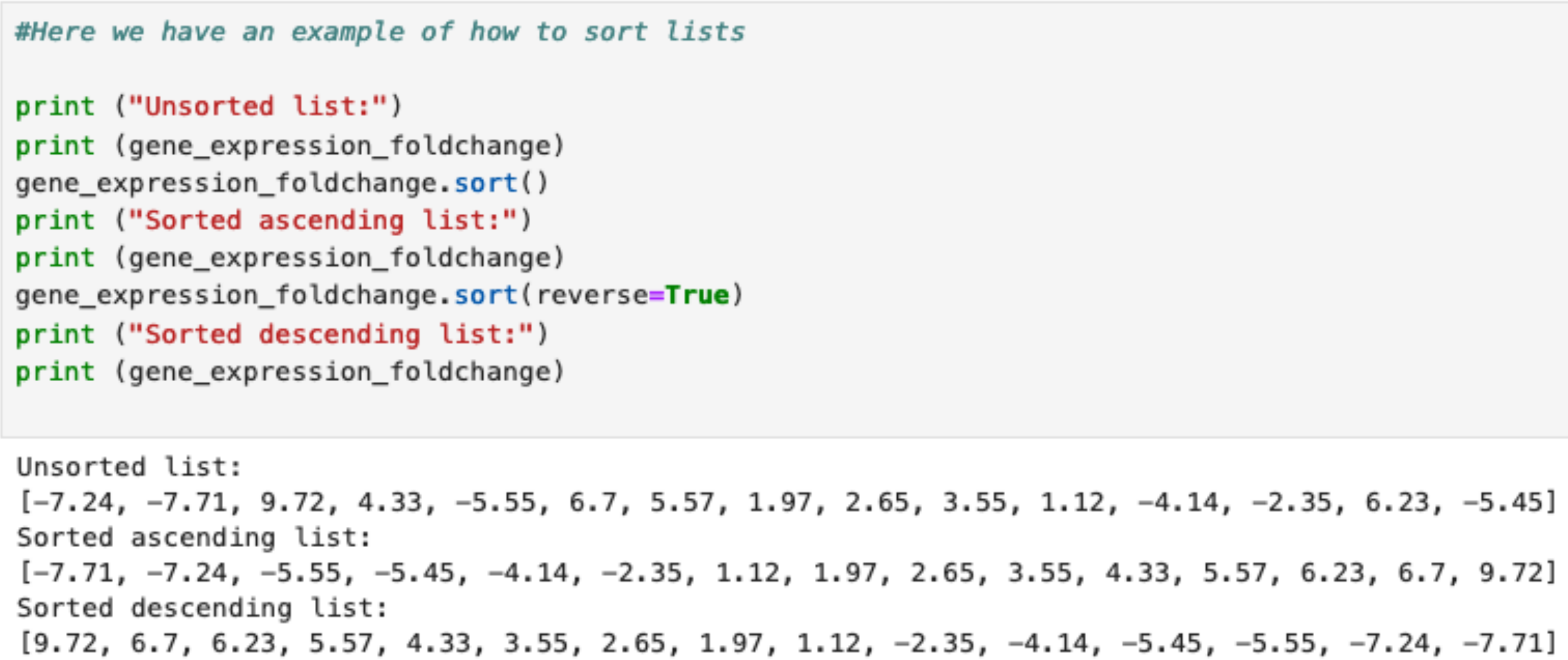
```
#Here we have an example of how to use the append method
import random

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.



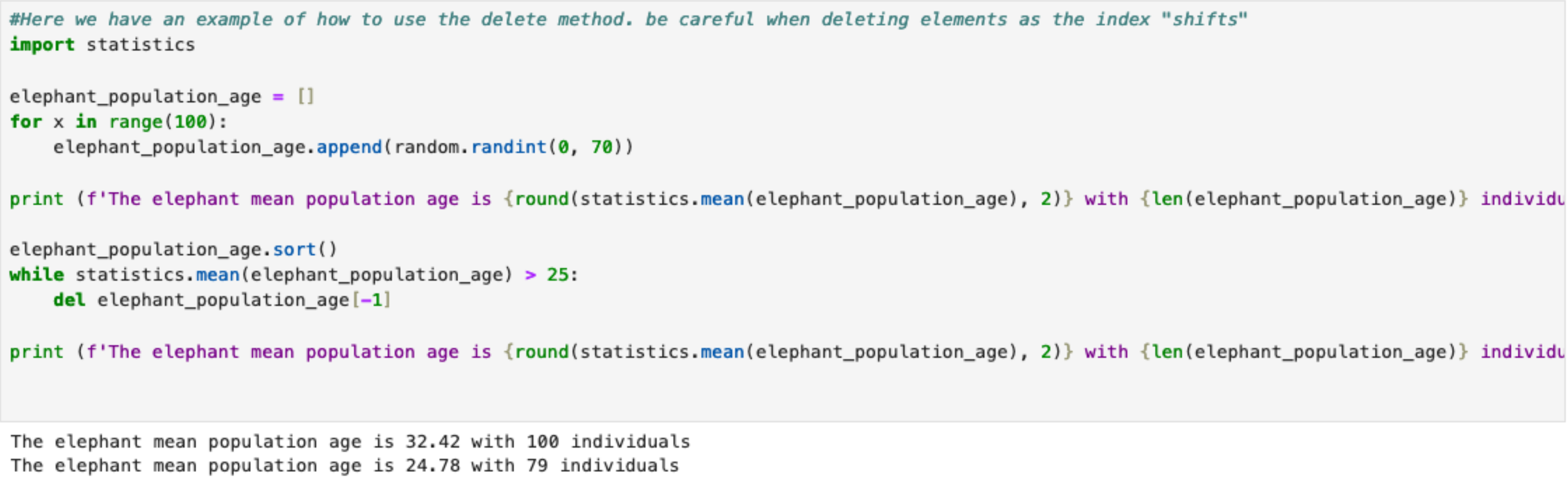
```
#Here we have an example of how to sort lists

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



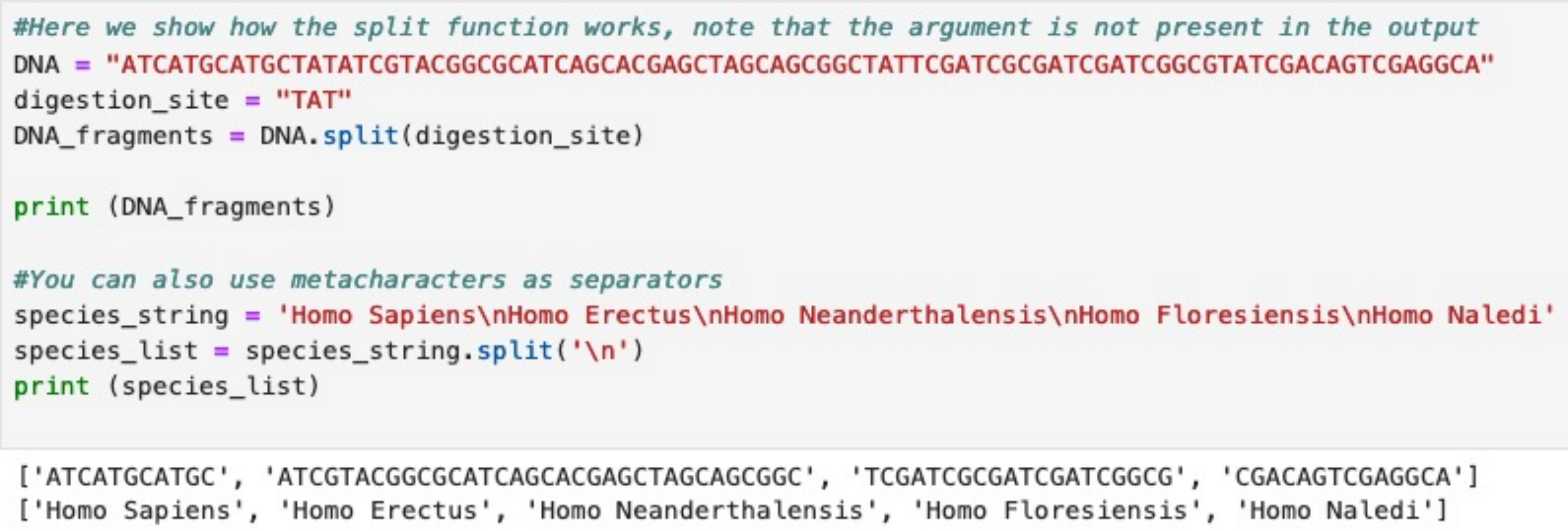
```
#Here we have an example of how to use the delete method. be careful when deleting elements as the index "shifts"
import statistics

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..)



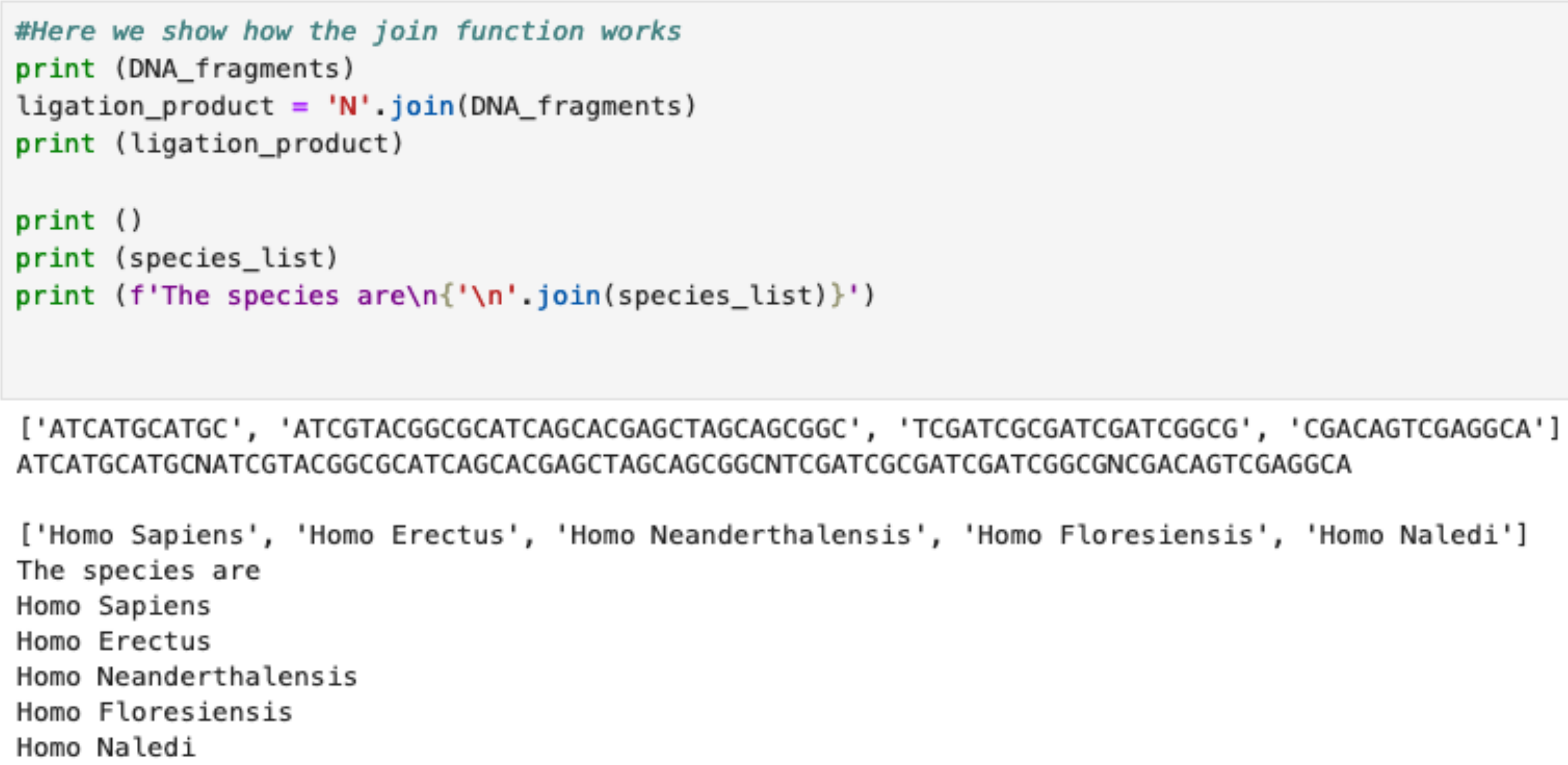
```
#Here we show how the split function works, note that the argument is not present in the output
DNA = "ATCATGCATGCTATATCGTACGGCGCATCAGCACGAGCTAGCAGCGGCTATTCGATCGCGATCGATCGGCGTATCGACAGTCGAGGCA"
digestion_site = "TAT"
DNA_fragments = DNA.split(digestion_site)

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 comprehension

An elegant solution to slim your code

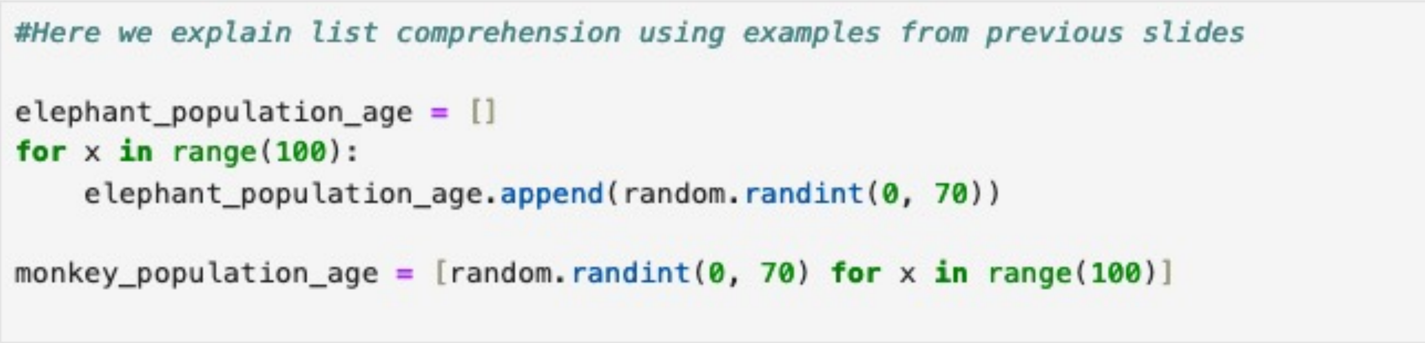
- List comprehension offers a shorter syntax when you want to create a new list

List comprehension syntax:

```
newlist = [expression for item in iterable]
```

Normal loop syntax:

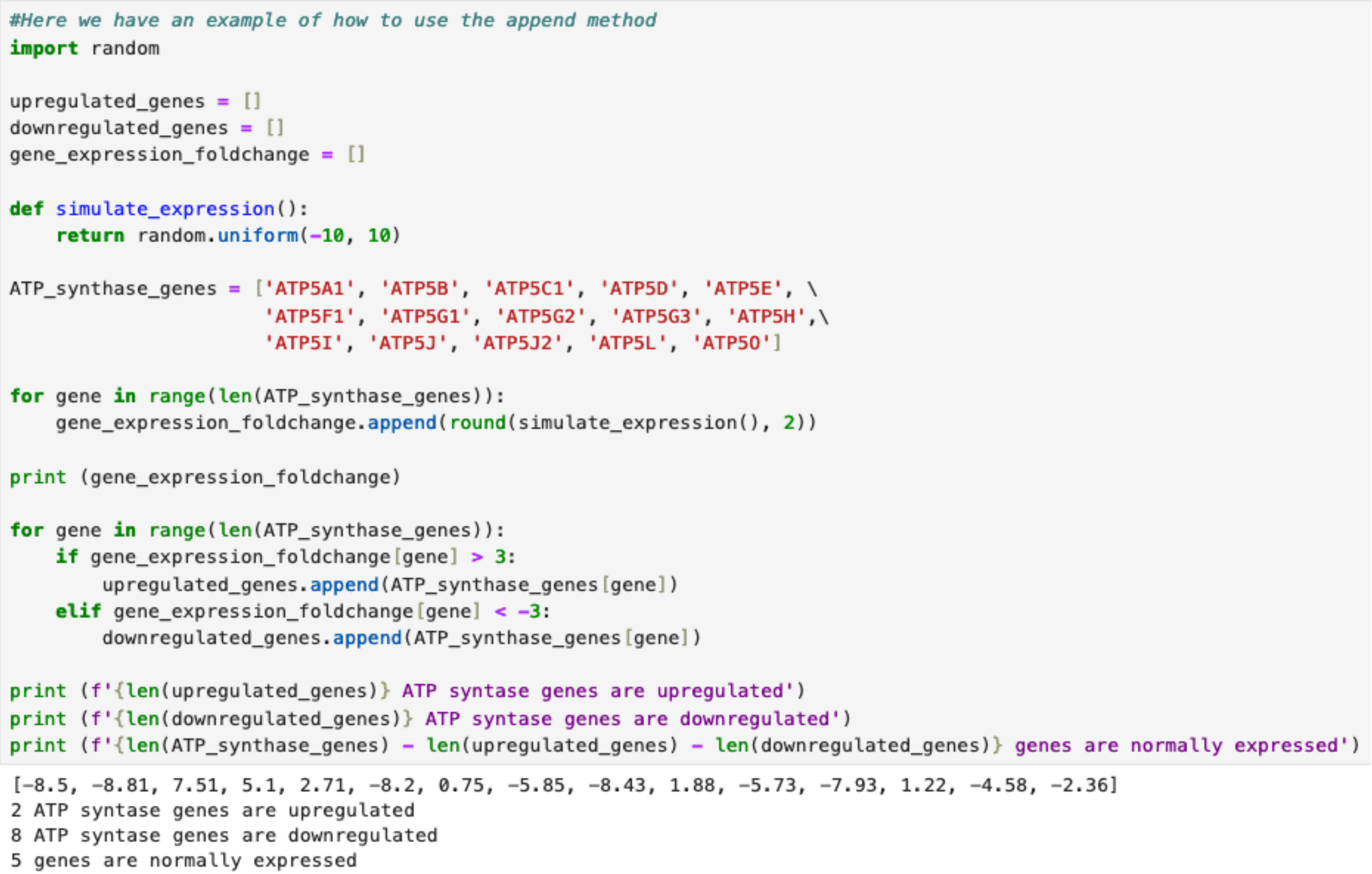
```
newList = []
for item in iterable:
   newList.append(expression)
```



```
#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)]
```

Hot question

Make a list comprehension for `gene_expression_foldchange`



```python
#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
```

CEPLAS
Cluster of Excellence on Plant Sciences

Introduction to Python

Break – 10 minutes

List comprehension with if/else conditions

Complex list comprehensions syntax

- In addition to simple list comprehension synthax it's possible to include if/else statements to shorten elaborate loops
- When the list comprehension becomes too complicated it's best to write out the loop normally for clarity

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]
```

List comprehensions with if/else conditions

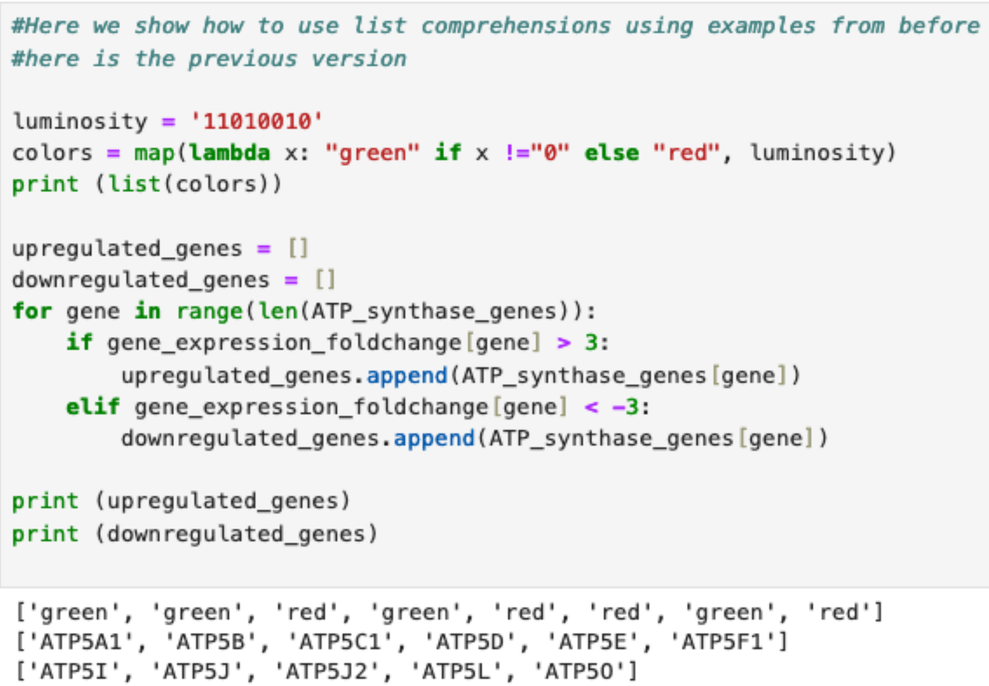
Examples

List comprehension synthax with if/else statements:

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

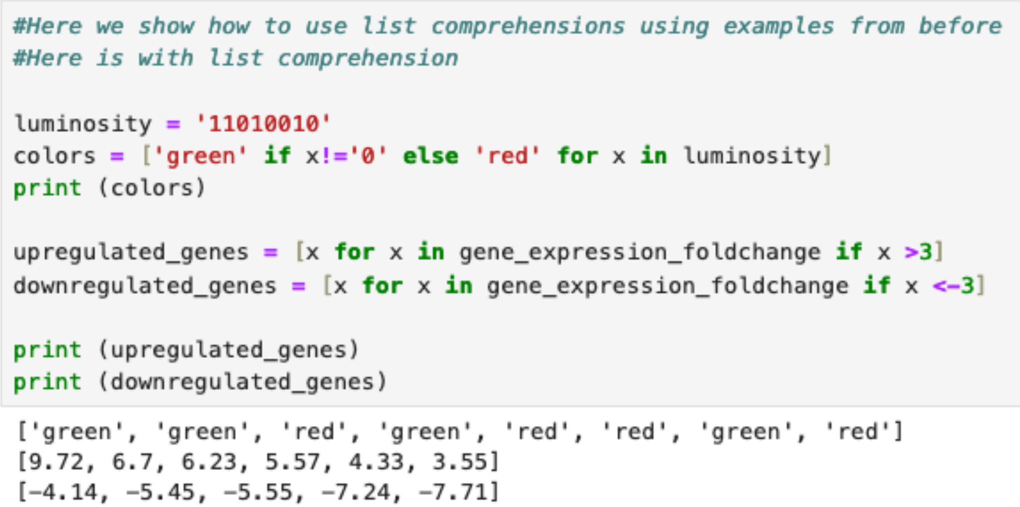
List comprehension synthax with **only** if statement:

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



```
#Here we 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']
```



```
#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]
```

Tuples

Data type for immutable sequences

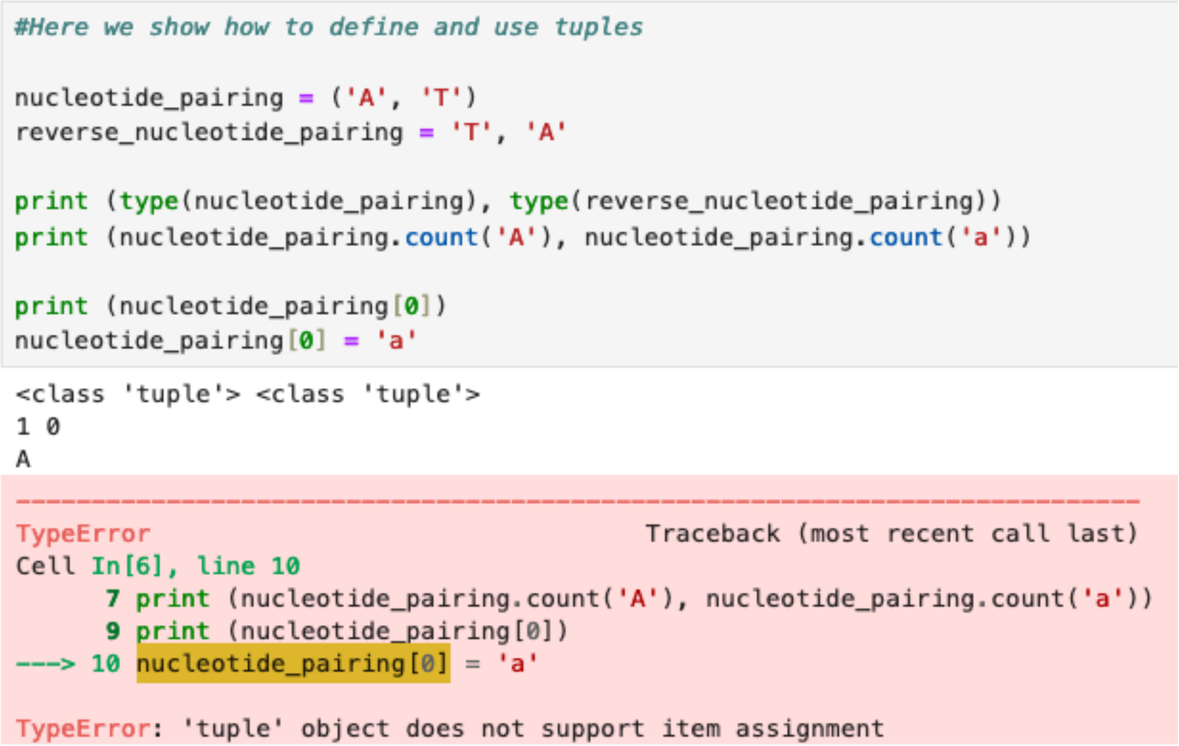
- A tuple is basically an immutable list
- To define tuples use either synthax:

```
Tuple_variable =(Variable1, Variable2, …)
Tuple_variable = Variable1, Variable2, …
```

- To define a tuple consisting only of one element you still have to add a comma at the end `(Variable,)` or `Variable,`
- Since tuples are sequences, we can use sequence-related functions – e.g. indexing, slicing, concatenation, multiplication, getting the min, max value

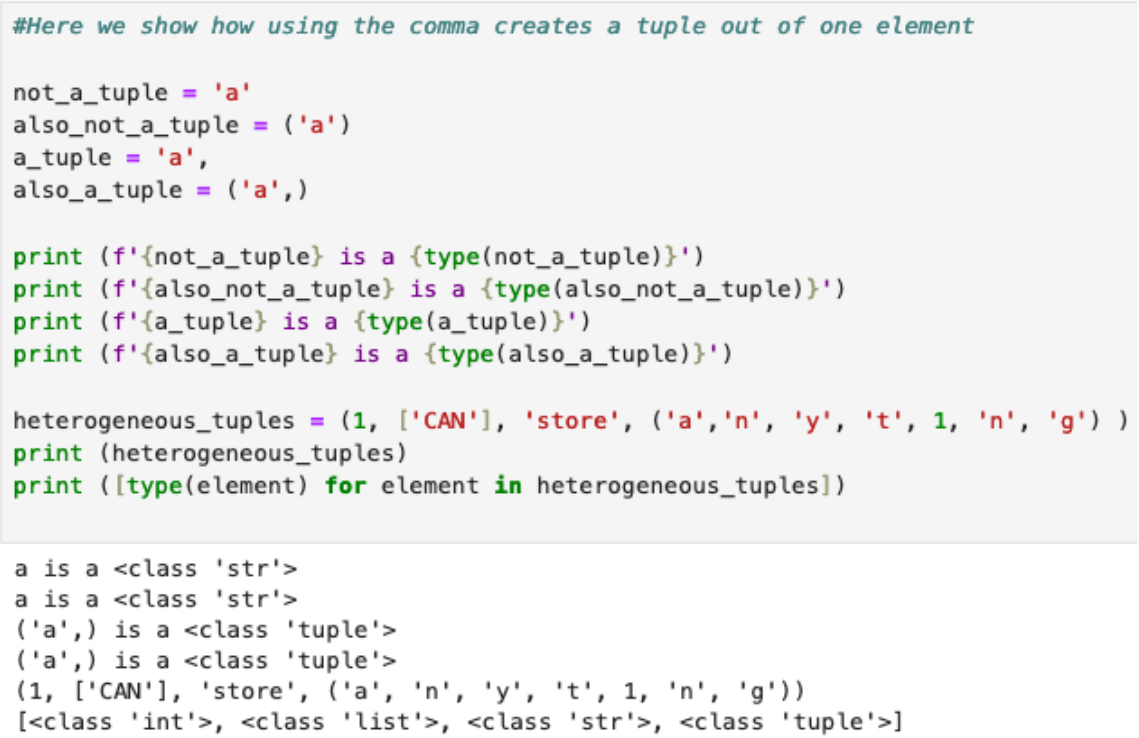
Tuples

Examples



```
#Here we show how to define and use tuples

nucleotide_pairing = ('A', 'T')
reverse_nucleotide_pairing = 'T', 'A'

print (type(nucleotide_pairing), type(reverse_nucleotide_pairing))
print (nucleotide_pairing.count('A'), nucleotide_pairing.count('a'))

print (nucleotide_pairing[0])
nucleotide_pairing[0] = 'a'
```

```
<class 'tuple'> <class 'tuple'>
1 0
A
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 10
      7 print (nucleotide_pairing.count('A'), nucleotide_pairing.count('a'))
      9 print (nucleotide_pairing[0])
---> 10 nucleotide_pairing[0] = 'a'

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



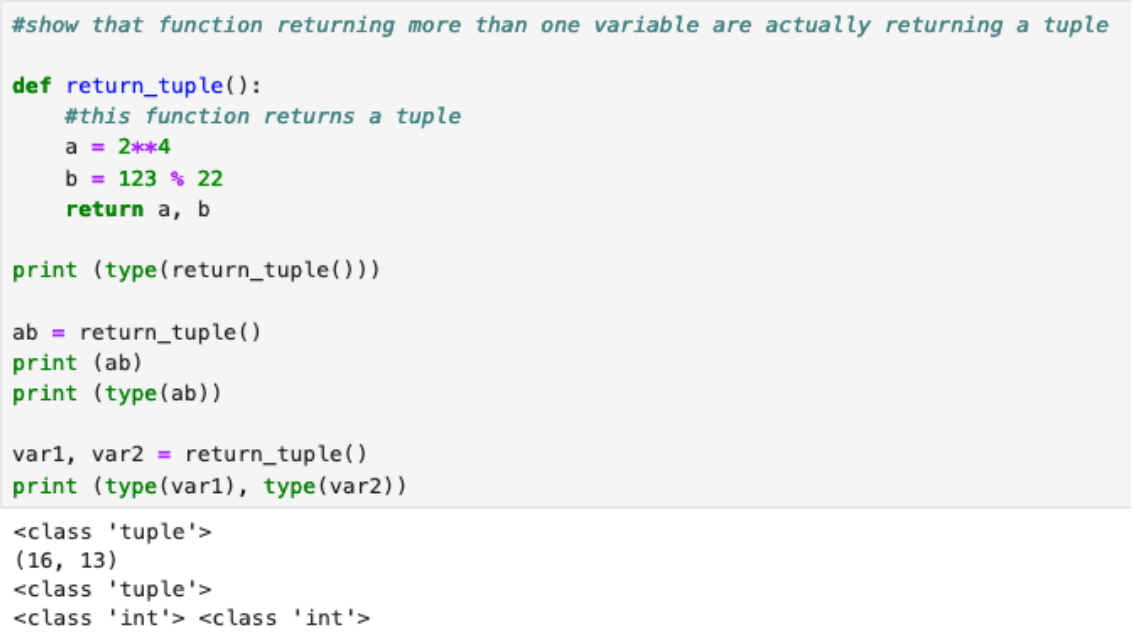
```
#Here we show how using the comma creates a tuple out of one element

not_a_tuple = 'a'
also_not_a_tuple = ('a')
a_tuple = 'a',
also_a_tuple = ('a',)

print (f'{not_a_tuple} is a {type(not_a_tuple)}')
print (f'{also_not_a_tuple} is a {type(also_not_a_tuple)}')
print (f'{a_tuple} is a {type(a_tuple)}')
print (f'{also_a_tuple} is a {type(also_a_tuple)}')

heterogeneous_tuples = (1, ['CAN'], 'store', ('a','n', 'y', 't', 1, 'n', 'g') )
print (heterogeneous_tuples)
print ([type(element) for element in heterogeneous_tuples])
```

```
a is a <class 'str'>
a is a <class 'str'>
('a',) is a <class 'tuple'>
('a',) is a <class 'tuple'>
(1, ['CAN'], 'store', ('a', 'n', 'y', 't', 1, 'n', 'g'))
[<class 'int'>, <class 'list'>, <class 'str'>, <class 'tuple'>]
```

What are tuples for?

Why shouldn’t I use a list instead?

- In most cases, you should use a list!
- Tuples are considered heterogeneous data structures while for list there is an expectation of homogeneity
- The fact that they are immutable makes them “reliable”, if you create something as a tuple, you know that you cannot “alter it” by mistake unless you assign a different tuple to the same variable name
- You have been using tuples already! When returning more than one value from a function, the variables are returned as a tuple (that you usually assign to individual variables)



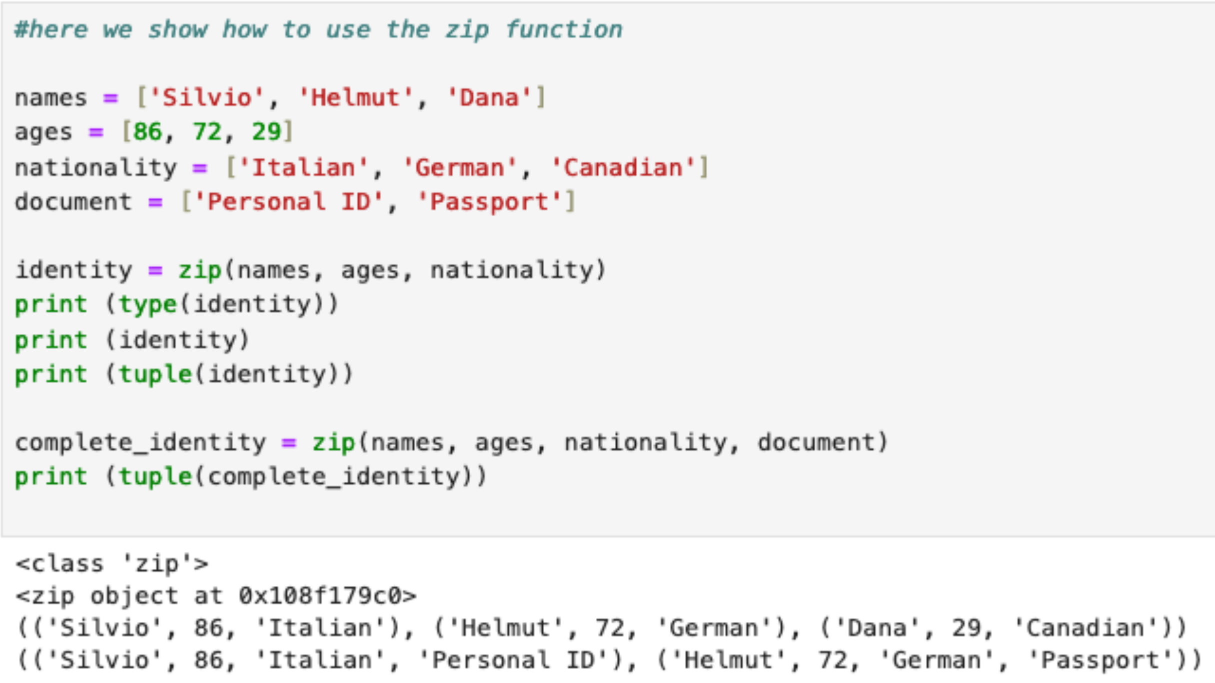
```
#show that function returning more than one variable are actually returning a tuple

def return_tuple():
    #this function returns a tuple
    a = 2**4
    b = 123 % 22
    return a, b

print (type(return_tuple()))

ab = return_tuple()
print (ab)
print (type(ab))

var1, var2 = return_tuple()
print (type(var1), type(var2))
```

```
<class 'tuple'>
(16, 13)
<class 'tuple'>
<class 'int'> <class 'int'>
```

`zip(…)` function

A function that returns a zip object, an iterator of tuples

- The `zip(…)` function takes the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc
- If the iterators are not of the same size, the function returns the zip object up to the shortest iterator



```
#here we show how to use the zip function

names = ['Silvio', 'Helmut', 'Dana']
ages = [86, 72, 29]
nationality = ['Italian', 'German', 'Canadian']
document = ['Personal ID', 'Passport']

identity = zip(names, ages, nationality)
print (type(identity))
print (identity)
print (tuple(identity))

complete_identity = zip(names, ages, nationality, document)
print (tuple(complete_identity))
```

```
<class 'zip'>
<zip object at 0x108f179c0>
(('Silvio', 86, 'Italian'), ('Helmut', 72, 'German'), ('Dana', 29, 'Canadian'))
(('Silvio', 86, 'Italian', 'Personal ID'), ('Helmut', 72, 'German', 'Passport'))
```

CEPLAS
Cluster of Excellence on Plant Sciences

Question time & Recap

RECAP week 2

Logical connectives

for-loops

while-loops

Writing better functions

Lambda and map

Local and global variables

Append, sort, delete, split

List comprehensions

Tuples and zip