Introduction to Bioinformatics using Python

Lecture 5: Python collections

Dr. Alexey Larionov

29 October 2024

www.cranfield.ac.uk

Learning outcomes

At the end of this lecture, you should be able to:

- Understand **list** and its operations
- Understand **tuple** and its operations
- Understand **dictionaries** and its operations
- Understand **set** and its operations

Learning outcomes

At the end of this lecture, you should be able to:

- Understand **list** and its operations
- Understand **tuple** and its operations
- Understand **dictionaries** and its operations
- Understand **set** and its operations

List

```
[ 42, 42, "September", 3.14 ]
```
Lists may even contain lists:
```
[ 42, "September", 3.14, ["A", "T", "C", "G"] ]
```

- **List** is a mutable collection of items, which can be changed in place
- Allows duplicate members, allows members of different types
- It is designated by square brackets

```
nucleotides = ['A', 'T', 'G', 'C', 'U']
print(nucleotides)

integers = [1, 2, 3, 4, 5]
print(integers)

floats = [3.14, 2.76, 9.99]
print(floats)
```

You can also make an empty list

```
empty = []
```

Let's make a list of (human) autosomes

```
autosomes = ["1", "2", "3", "4", "5",
"6", "7", "8", "9", "10", "11", "12",
"13", "14", "15", "16", "17", "18",
"19", "20", "21", "22"]
print(autosomes)
```

There is an easier way of creating such list:

```
autosomes = list(range(1,22))
print(autosomes)
```

What are the differences?

List operations

Similar to Strings operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

```
# concatenate two lists
autos1 = list(range(1,10))
autos2 = list(range(11,23))
autosomes = autos1 + autos2
print(autosomes)
```

```
# repeat a list
more_autosomes = autos2 * 3
print(more_autosomes)
```

```
# access a list item, say 3rd item
third_autosome = autosomes[2]
print(third_autosome)
```

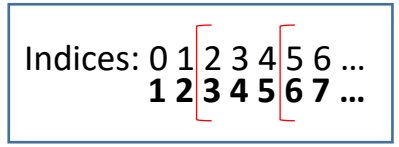
3

```
# length of the list
length = len(autosomes)
print(length)
```

21

```
# slicing autosomes
sliced_autosomes = autosomes[2:5]
print(sliced_autosomes)
```

[3, 4, 5]



List operations

Similar to Strings operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

```
# concatenate two lists
autos1 = list(range(1,10))
autos2 = list(range(11,23))
autosomes = autos1 + autos2
print(autosomes)
```

```
# repeat a list
more_autosomes = autos2 * 3
print(more_autosomes)
```

```
# access a list item, say 3rd item
third_autosome = autosomes[2]
print(third_autosome)
```

3

length() In R

```
# length of the list
length = len(autosomes)
print(length)
```

21 → Why not 22 ?

```
# slicing autosomes
sliced_autosomes = autosomes[2:5]
print(sliced_autosomes)
```

[3, 4, 5]

Indices: 0 1 2 3 4 5 6 …
1 2 3 4 5 6 7 …

List operations

Similar to Strings operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

```
# concatenate two lists
autos1 = list(range(1,10))
autos2 = list(range(11,23))
autosomes = autos1 + autos2
print(autosomes)
```

Creates 1 to 9
NOT
1 to 10

```
# repeat a list
more_autosomes = autos2 * 3
print(more_autosomes)
```

```
# access a list item, say 3rd item
third_autosome = autosomes[2]
print(third_autosome)
```

3

length()
In R

```
# length of the list
length = len(autosomes)
print(length)
```

21 → Why not 22 ?

```
# slicing autosomes
sliced_autosomes = autosomes[2:5]
print(sliced_autosomes)
```

[3, 4, 5]

Indices: 0 1 2 3 4 5 6 …
1 2 3 4 5 6 7 …

List methods

Operations that were not available for strings

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

```
autosomes = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22]
```

```
# add a new element to the list
autosomes.append(10)
print(autosomes)

# sort the elements
autosomes.sort()
print(autosomes)

# reverse your new list
autosomes.reverse()
print(autosomes)

# find the index of chromosome 5
chr5_index = autosomes.index(5)
print("chr5 is at: ", chr5_index)

# find how many times chr10 is in the list
chr10_count = autosomes.count(10)

# remove a chromosome
autosomes.remove(17)
```

List methods

Operations that were not available for strings

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

```
autosomes = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10
11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22]
```

```
# add a new element to the list
autosomes.append(10)
print(autosomes)
```

Wrong for lists !

```
full_autosomes = autosomes.append(10)
```

```
# sort the elements
autosomes.sort()
print(autosomes)

# reverse your new list
autosomes.reverse()
print(autosomes)

# find the index of chromosome 5
chr5_index = autosomes.index(5)
print("chr5 is at: ", chr5_index)

# find how many times chr10 is in the list
chr10_count = autosomes.count(10)

# remove a chromosome
autosomes.remove(17)
```

List methods *etc*

Operations that were not available for strings

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

List comprehension	https://realpython.com/list-comprehension-python

```
x = [ n/2 for n in range(1:10) if n < 5 ]
print(x)
```

```
[1.0, 1.5, 2.0]
```

```
# add a new element to the list
autosomes.append(10)
print(autosomes)
```

Wrong for lists !

```
full_autosomes = autosomes.append(10)
```

```
# sort the elements
autosomes.sort()
print(autosomes)
```

```
# reverse your new list
autosomes.reverse()
print(autosomes)
```

```
# find the index of chromosome 5
chr5_index = autosomes.index(5)
print("chr5 is at: ", chr5_index)
```

```
# find how many times chr10 is in the list
chr10_count = autosomes.count(10)
```

```
# remove a chromosome
autosomes.remove(17)
```

+ *vs* extend *vs* append

- **+** creates a new list in memory - with new id()

```
a_list = [1,2,3,4,5,6]
new_list  = [7, 8, 9]
a_new_list = a_list + new_list
print(a_new_list)
```

- **extend** operates on list in place

```
a_list = [1,2,3,4,5,6]
a_list.extend([7, 8, 9])
print(a_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **append** adds a ***singleton*** into list

```
a_list = [1,2,3,4,5,6]
a_list.append ([7, 8, 9])
print(a_list)
```

```
[1, 2, 3, 4, 5, 6,[7, 8, 9]]
```

+ *vs* extend *vs* append

- **+** creates a new list in memory - with new id()

```
a_list = [1,2,3,4,5,6]
new_list  = [7, 8, 9]
a_new_list = a_list + new_list
print(a_new_list)
```

- **extend** operates on list in place

```
a_list = [1,2,3,4,5,6]
a_list.extend([7, 8, 9])
print(a_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **append** adds a ***singleton*** into list

```
a_list = [1,2,3,4,5,6]
a_list.append ([7, 8, 9])
print(a_list)
```

```
[1, 2, 3, 4, 5, 6,[7, 8, 9]]
```

Don't use assignment for methods that operate in-place:

```
another_list = a_list.append ([7,8,9])
print(another_list)
```

```
None
```

append & extend operate <u>in place</u>
so, they don't *return* any values !

Learning outcomes

At the end of this lecture, you should be able to:

- Understand **list** and its operations
- Understand **tuple** and its operations
- Understand **dictionaries** and its operations
- Understand **set** and its operations

Tuples

- Tuple, like list is a collection of items
- *Tuples* are *immutable* like strings

The comma is the tuple creation operator, not parentheses

```
>>> 1,
(1,)
```

Python shows parentheses for clarity (best practice, “syntactic sugar”)

```
>>> (1,)
(1,)
```

Don't forget the comma!

```
>>> (1)
1
```

Empty tuples don’t need comma: they have a special syntactic form

```
>>> () or tuple()
()
```

Trailing comma required only for singletons: (1,) but (1,2,3)

Tuples vs. Lists

Content of this list CAN be changed in place; however, the list cannot be removed from the tuple or changed to a different data type in the tuple

- Like the lists, tuples can contain elements of **different data types**

```
( 42 , 3.14 , "September", (1,2,3,4), ["A", "B"] )
```

- Lists have **more features** than tuples, but are **slower** & take **more memory**
  - Lists can be modified, and they have lots of handy operations and methods
  - Tuples are immutable, have fewer features (but faster and smaller in memory :)

- To convert between tuples and lists use the **`list()`** and **`tuple()`** functions:

```
a_list = list(a_tuple)
a_tuple = tuple(a_list)
```

Tuple operations

- Many things work like in lists, e.g.:

Length of a tuple:

```
length = len(autosomes)
print(length)
```

etc:

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

```
autosomes = tuple(range(1,23))
```

- Sort chromosomes in tuple ?

```
autosomes.sort()
```

Traceback (most recent call last):
…
AttributeError:
'tuple' object has no attribute 'sort'

- An alternative:

```
sorted_autosomes = sorted(autosomes)
print(sorted_autosomes)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22]

What is the output data type?

Tuples and Lists – Mutability revision

List

```
nucleotides = ["A", "T", "G", "C"]
print(nucleotides[1])     "T"

nucleotides[1] = "U"
print(nucleotides)
```

```
["A", "U", "G", "C"]
```

Tuple

```
nucleotides = ("A", "T", "G", "C")
print(nucleotides[1])     "T"

nucleotides[1] = "U"
```

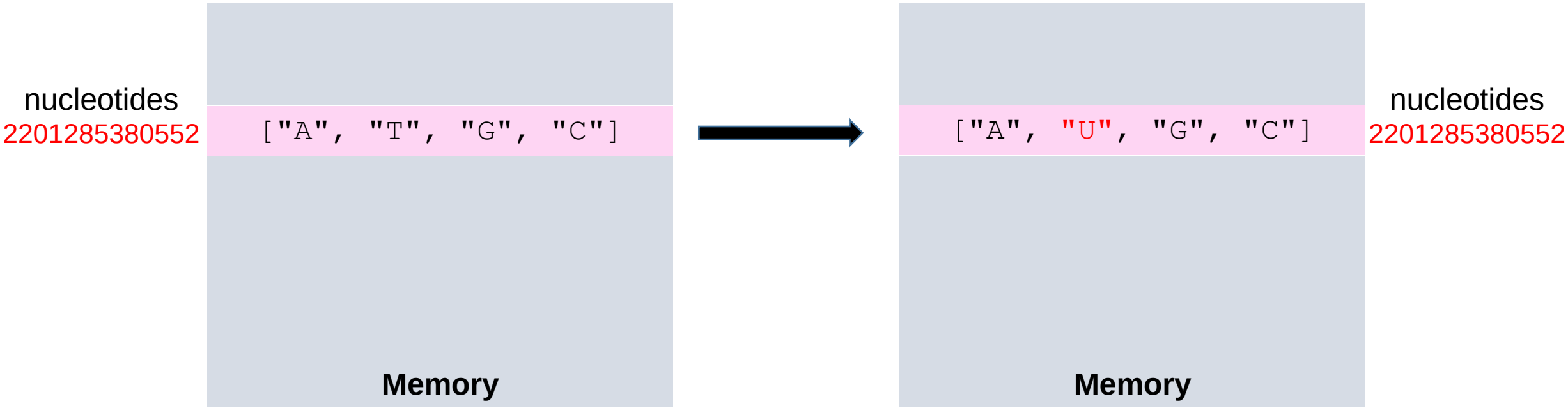
```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
support item assignment
```

Solution:

```
nucleotides = ("A", "U", "G", "C")
print(nucleotides)
```

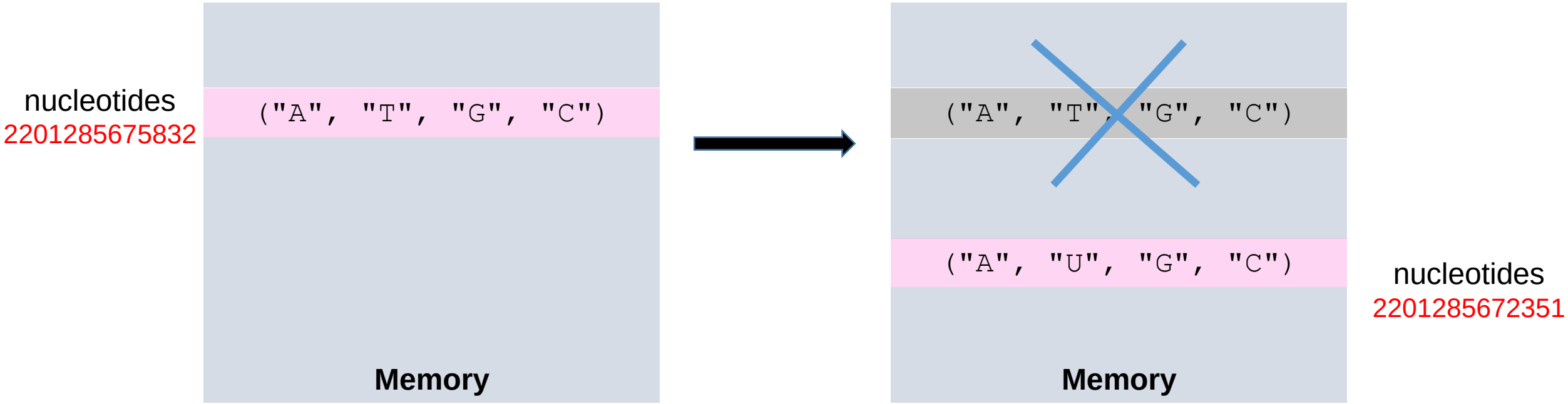
Mutable vs. Immutable : List

```
nucleotides = ["A", "T", "G", "C"]
print(id(nucleotides))
nucleotides[1] = "U"
print(id(nucleotides))
```

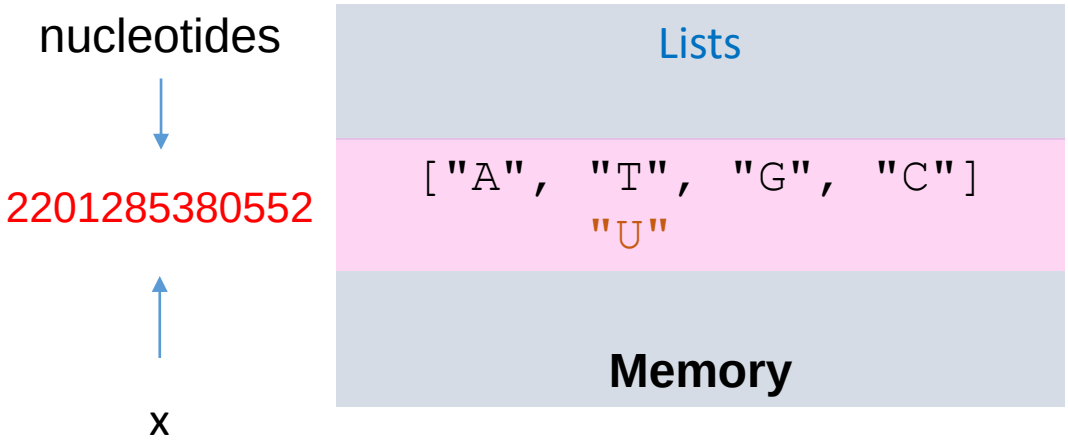


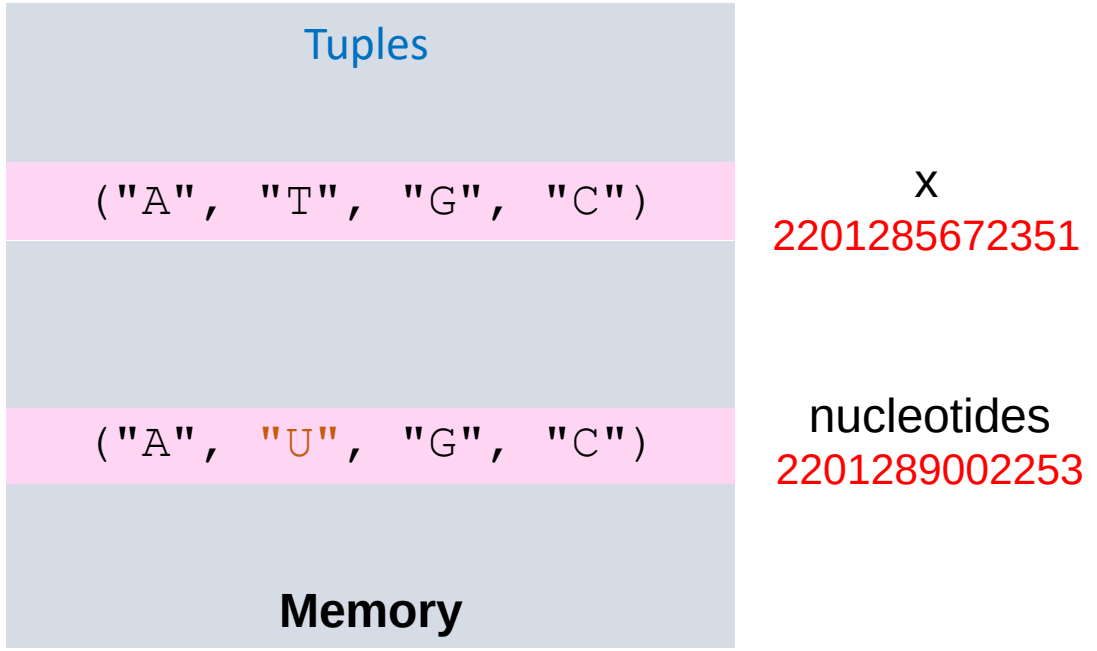
Mutable vs. Immutable : Tuple

```
nucleotides = ("A", "T", "G", "C")
print(id(nucleotides))
nucleotides = ("A", "U", "G", "C")
print(id(nucleotides))
```



Mutable vs. Immutable
(variables and memory objects)






```
nucleotides = ["A", "T", "G", "C"]
id(nucleotides)    # …0552
x = nucleotides
id(x)              # …0552

nucleotides[1] = "U"
id(nucleotides)    # …0552
id(x)              # …0552
```

```
nucleotides = ("A", "T", "G", "C")
id(nucleotides)    # …2351
x = nucleotides
id(x)              # …2351

nucleotides = ("A", "U", "G", "C")
id(nucleotides)    # …2253
id(x)              # …2351
```

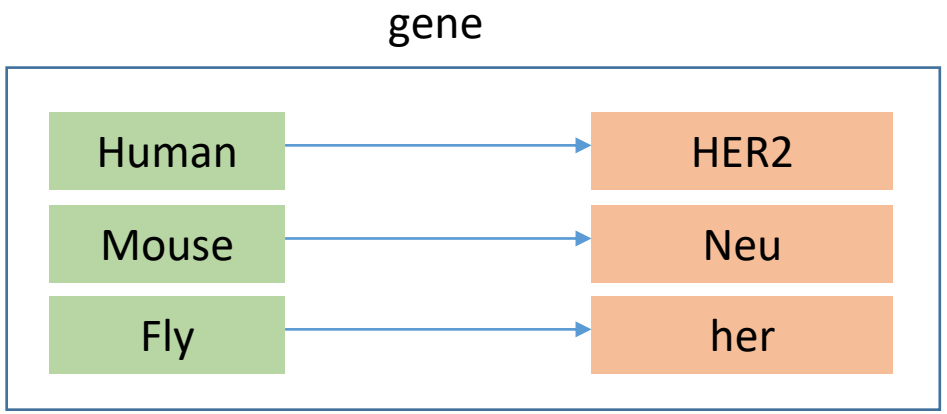
Learning outcomes

At the end of this lecture, you should be able to:

- ✓ Understand **list** and its operations
- ✓ Understand **tuple** and its operations
- Understand **dictionaries** and its operations
- Understand **set** and its operations

Dictionaries

- Store ***pairs*** of entries (called ***items)***
- Each pair of entries contains
  - A ***key*** – *No duplicates !*
  - *A* ***value*** – *Any data type*



- Dictionary is enclosed within curly brackets
- Paris of entries are separated by commas
- Key and value are separated by a colon

```
gene = {'Human' : 'HER2', 'Mouse' : 'Neu', 'Fly' : 'her'}
```

https://realpython.com/iterate-through-dictionary-python

https://realpython.com/lessons/iterate-through-dictionary-python-overview

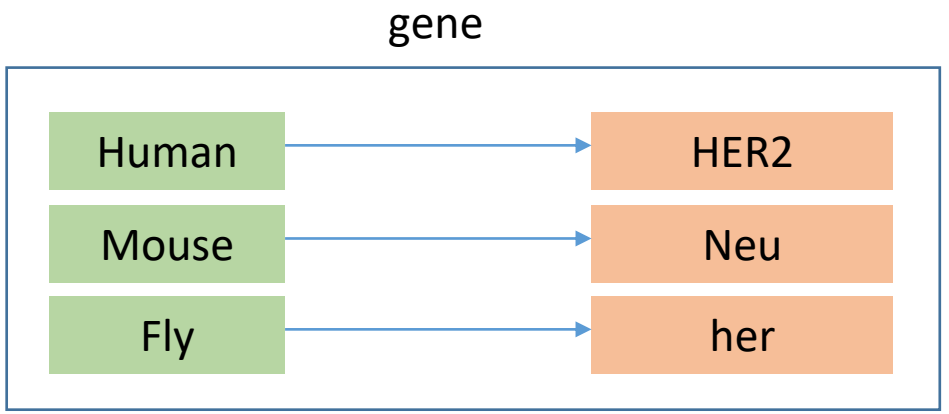
Dictionaries

- Store ***pairs*** of entries (called ***items)***
- Each pair of entries contains
  - A ***key*** – *No duplicates !*
  - *A* ***value*** – *Any data type*

Keys allow only "hashable" (~ immutable) data type:
use strings, numbers or tuples if uncertain

```
{"a":"Hello!","b":(1,2,3)}          OK
{(3.14,42):"Hello!","b":(1,2,3)}    OK
{[3.14,42]:"Hello!","b":(1,2,3)} TypeError:
                                 unhashable type
```

- Dictionary is enclosed within curly brackets
- Paris of entries are separated by commas
- Key and value are separated by a colon




```
gene = {'Human' : 'HER2', 'Mouse' : 'Neu', 'Fly' : 'her'}
```

https://realpython.com/iterate-through-dictionary-python

https://realpython.com/lessons/iterate-through-dictionary-python-overview

Using dictionary

Dictionary values are accessed by its *keys*

- If we have

```
gene = {'Human' : 'HER2', 'Mouse' : 'Neu', 'Fly' : 'her'}
```

then

```
gene['Human'] is 'HER2'
```

and

```
gene['Mouse'] is 'Neu'
```

and

```
gene['Fly'] is 'her'
```

Using dictionary

Dictionary values are accessed by its *keys*

- If we have

```
gene = {'Human' : 'HER2', 'Mouse' : 'Neu', 'Fly' : 'her'}
```

then

```
gene['Human'] is 'HER2'
```

and

```
gene['Mouse'] is 'Neu'
```

and

```
gene['Fly'] is 'her'
```

What happens if we try to access a non-existing key?

```
gene['worm']
Traceback (most recent call last):
  File "<ipython-input-53-51549dfb1e7a>",
line 1, in <module>
    gene['worm']
KeyError: 'worm'
```

Dictionaries are mutable

```
gene = {'Human' : 'HER2', 'Mouse' : 'Neu', 'Fly' : 'her'}
print(gene)
id(gene)

# If we assign a new value to a key, or add/ remove key-value pair
gene['Human'] = 'ERBB2'
del(gene['Fly'])

# Then the dictionary is it updated in place (i.e. its position in the memory is still the same)
print(gene)
id(gene)
```

```
Output:    {'Human': 'HER2', 'Mouse': 'Neu', 'Fly': 'her'}
           1791452545256
           {'Human': 'ERBB2', 'Mouse': 'Neu'}
           1791452545256
```

Dictionary methods

Python is constantly evolving …

Method	Meaning
~~<dict>.has_key(<key>)~~ **<key> in <dict>**	Returns true if dictionary contains the specified key, false if it doesn't. **(has_key removed in Python 3)**
<dict>.keys()	Returns a list* of the keys.
<dict>.values()	Returns a list* of the values.
<dict>.items()	Returns a list* of tuples (key,value) representing the key-value pairs.
del <dict>[<key>]	Delete the specified entry.
<dict>.clear()	Delete all entries.

* ***Lists*** were returned in Python 2,
Python 3 returns special types of ***iterable objects,*** which can be converted to lists)

Dictionary methods

```
gene =
{'Human':'HER2', 'Mouse':'Neu', 'Fly':'her'}
```

- **Adding to** the dictionary

```
gene['Worm'] = 'her'
print(gene)
```

```
{'Human':'HER2', 'Mouse':'Neu',
'Fly':'her', 'Worm':'her'}
```

- **Removing from** the dictionary

```
del(gene['mouse'])
print(gene)
```

```
{'Human':'HER2', 'Fly':'her',
'Worm':her'}
```

- **Checking** if a specific **key** is present

```
'Human' in gene.keys()   True
'Human' in gene          True

'Human ' in gene           ?
'mouse' in gene            ?
```

Dictionary methods

```
gene =
{'Human':'HER2', 'Mouse':'Neu', 'Fly':'her'}
```

- **Extract keys and values** from a dictionary

```
items = gene.items()
print(items)
```

```
dict_items([('Human', 'HER2'), ('Mouse', 'Neu'), ('Fly', 'her')])
```

- **Extract the keys** from a dictionary

```
keys = gene.keys()
print(keys)
```

```
dict_keys(['Human', 'Mouse', 'Fly'])
```

- **Extract the values** from a dictionary

```
values = gene.values()
print(values)
```

```
dict_values(['HER2', 'Neu', 'her'])
```

Note that these functions return special iterable data types (can be converted to lists if needed)

More dictionary methods

Python Dictionary Methods	
Method	**Description**
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and values
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Learning outcomes

At the end of this lecture, you should be able to:

- ✓ Understand **list** and its operations
- ✓ Understand **tuple** and its operations
- ✓ Understand **dictionaries** and its operations
- Understand **set** and its operations

Sets: *unordered* collections of *unique* elements

- Can only contain ***unique*** elements ***- Duplicates are eliminated***

```
dnabases = {'A', 'T', 'G', 'C', 'A'}
print(dnabases) # {'A', 'T', 'G', 'C'}
```

- Identified by ***curly brackets***

```
nucleotides = {'A', 'T', 'G', 'C', 'U'}
print(nucleobases)
dnabases = {'A', 'T', 'G', 'C'}
print(dnabases)
rnabases = {'A', 'U', 'G', 'C'}
print(rnabases)
```

- Sets are ***not ordered***

```
{1,2,3} == {3,2,1} # True (set)
[1,2,3] == [3,2,1] # False (list)
```

so, sets don't support indexing:

```
dnabases[0]          # Error
```

- While sets themselves are ***mutable***, they only can contain ***"hashable" (immutable) elements***
  (e.g. set may contain numbers, tuples and strings, but can NOT contain lists or dictionaries)

```
curiosity_set = {1, 3.14, (1,2,3), 'Hi'}
```

There are "Frozen sets" that are immutable

Set algebraic operations

- **Set Union**: `set1.union(set2)`

  A new set with the elements of both *set1* and *set2*

  ```
  bases = dnabases.union(rnabases)
  print(bases)          {'C', 'T', 'A', 'U', 'G'}
  ```

  `set1 | set2`

- **Set Intersection**: `set1.intersection(set2)`

  A new set with the elements that are common to *set1* and *set2*

  ```
  common_bases = dnabases.intersection(rnabases)
  print(common_bases)     {'G', 'C', 'A'}
  ```

  `set1 & set2`

- **Set Difference**: `set1.difference(set2)`

  A new set with the elements in *set1* that are not in *set2*

  ```
  bases = nucleotides.difference(dnabases)
  print(bases)                    {'U'}
  ```

  `set1 - set2`

Set comparison operations

- **Disjoint set**: `set1.isdisjoint(set2)`

  True if the set and the argument have no elements in common

  ```
  dnabases.isdisjoint(rnabases)
  ```

  FALSE

- **Subset**: `set1.issubset(set2)`

  True if every element of *set1* is also in *set2*

  ```
  dnabases.issubset(nucleotides)
  ```

  ```
  set1 <= set2
  set1 < set2
  ```

  TRUE

- **Superset**: `set1.issuperset(set2)`

  True if every element of *set2* is also in set1

  ```
  nucleotides.issuperset(dnabases)
  ```

  ```
  set1 >= set2
  set1 > set2
  ```

  TRUE

More set methods

Python Sets Methods	
Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Learning outcomes

At the end of this lecture, you should be able to:

- Understand **list** and its operations
- Understand **tuple** and its operations
- Understand **dictionaries** and its operations
- Understand **set** and its operations

Questions