Download Chapter pdf

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

- How is a dictionary defined in Python?
- What are the ways to interact with a dictionary?
- Can a dictionary be nested?

- Understanding the structure of a dictionary.
- Accessing data from a dictionary.
- Practising nested dictionaries to deal with complex data.

This chapter assumes that you are familiar with the following concepts in Python 3:

Prereq

- Indentation Rule
- Conditional Statements
- Arrays
- Loops and Iterations

Dictionary

Mapping Types – `dict`

Google search

StackOverflow python-3.x dictionaries

YouTube Tutorial Dictionaries

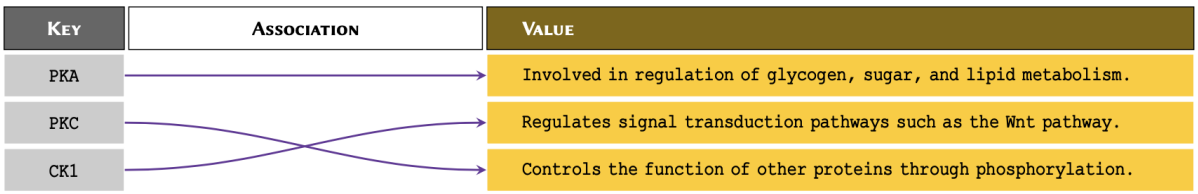
One of the most useful built-in tools in Python, dictionaries associate a set of *values* with a number of *keys*.

Think of an old fashion, paperback dictionary where we have a range of words with their definitions. The words are the *keys*, and the definitions are the *values* that are associated with the keys. A Python dictionary works in the same way.

Consider the following scenario:

> Suppose we have a number of protein kinases, and we would like to associate them with their descriptions for future reference.

This is an example of association in arrays. We may visualise this problem as displayed in Figure.



One way to associate the proteins with their definitions would be to use nested arrays. However, it would make it difficult to retrieve the values at a later time. This is because to retrieve the values, we would need to know the index at which a given protein is stored.

Instead of using normal arrays, in such circumstances, we use *associative arrays*. The most popular method to create construct an associative array in Python is to create dictionaries or `dict`.

Remember

To implement a `dict` in Python, we place our entries in curly bracket, separated using a comma. We separate *keys* and *values* using a colon — e.g. {'key': 'value'}. The combination of dictionary *key* and its associating *value* is known as a dictionary *item*.

Note

When constructing a long `dict` with several *items* that span over several lines, it is not necessary to write one *item* per line or use indentations for each *item* or line. All we must is to write the *items* as {'key': 'value'} in curly brackets and separate each pair with a comma. However, it is good practice to write one *item* per line and use indentations as it makes it considerably easier to read the code and understand the hierarchy.

We can therefore implement the diagram displayed in Figure in Python as follows:

```
1 protein_kinases = {
2   'PKA': 'Involved in regulation of glycogen, sugar, and lipid
       metabolism.',
3   'PKC': 'Regulates signal transduction pathways such as the Wnt
       pathway.',
4   'CK1': 'Controls the function of other proteins through
       phosphorylation.'
5   }
6
7 print(protein_kinases)
```

```
{'PKA': 'Involved in regulation of glycogen, sugar, and lipid
    metabolism.', 'PKC': 'Regulates signal transduction pathways such as
     the Wnt pathway.', 'CK1': 'Controls the function of other proteins
    through phosphorylation.'}
```

```
print(type(protein_kinases))
```

```
<class 'dict'>
```

Do it Yourself

Use Universal Protein Resource (UniProt) to find the following proteins for humans: - Axin-1 - Rhodopsin

Construct a dictionary for these proteins and the number amino acids for each of them. The *keys* should represent the name of the protein. Display the result.

```
proteins = {
  'Axin-1': 862,
  'Rhodopsin': 348
  }

print(proteins)
```

```
{'Axin-1': 862, 'Rhodopsin': 348}
```

Now that we have created a dictionary; we can test whether or not a specific *key* exists our dictionary:

```
'CK1' in protein_kinases
```

```
True
```

```
'GSK3' in protein_kinases
```

```
False
```

Do it Yourself

Using the dictionary you created in Do it Yourself, test to see whether or not a protein called **ERK** exists as a *key* in your dictionary? Display the result as a Boolean value.

```
print('ERK' in proteins)
```

```
False
```

Interacting with a dictionary

We have already learnt that in programming, the more explicit our code, the better it is. Interacting with dictionaries in Python is very easy, coherent, and explicit. This makes them a powerful tool that we can exploit for different purposes.

In arrays, specifically in `list` and `tuple`, we routinely use indexing techniques to retrieve *values*. In dictionaries, however, we use *keys* to do that. Because we can define the *keys* of a dictionary ourselves, we no longer have to rely exclusively on numeric indices.

As a result, we can retrieve the *values* of a dictionary using their respective *keys* as follows:

```
print(protein_kinases['CK1'])
```

```
Controls the function of other proteins through phosphorylation.
```

However, if we attempt to retrieve the *value* for a *key* that does not exist in our `dict`, a `KeyError` will be raised:

```
'GSK3' in protein_kinases
```

```
False
```

```
print(protein_kinases['GSK3'])
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: '
    GSK3'

Detailed traceback:
  File "<string>", line 1, in <module>
```

Do it Yourself

Implement a `dict` to represent the following set of information:

Cystic Fibrosis:

Full Name	Gene	Type
Cystic fibrosis transmembrane conductance regulator	CFTR	Membrane Protein

Using the dictionary you implemented, retrieve and display the *gene* associated with cystic fibrosis.

```
cystic_fibrosis = {
  'full name': 'Cystic fibrosis transmembrane conductance regulator',
  'gene': 'CFTR',
  'type': 'Membrane Protein'
  }

print(cystic_fibrosis['gene'])
```

```
CFTR
```

Remember

Whilst the *values* in a `dict` can be of virtually any type supported in Python, the *keys* may only be defined using immutable types.

To find out which types are immutable, see Table. Additionally, the *keys* in a dictionary must be unique.

If we attempt to construct a `dict` using a mutable value as *key*, a `TypeError` will be raised.

For instance, `list` is a mutable type and therefore cannot be used as a *key*:

```
test_dict = {
  ['a', 'b']: 'some value'
  }
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): TypeError:
    unhashable type: 'list'

Detailed traceback:
  File "<string>", line 1, in <module>
```

But we can use any immutable type as a *key*:

```
test_dict = {
  'ab': 'some value'
  }

print(test_dict)
```

```
{'ab': 'some value'}
```

```
test_dict = {
  ('a', 'b'): 'some value'
  }

print(test_dict)
```

```
{('a', 'b'): 'some value'}
```

If we define a *key* more than once, the Python interpreter constructs the entry in `dict` using the last instance.

In the following example, we repeat the *key* 'pathway' twice; and as expected, the interpreter only uses the last instance, which in this case represents the value 'Canonical':

```
signal = {
  'name': 'Wnt',
  'pathway': 'Non-Canonical',  # first instance
  'pathway': 'Canonical'  # second instance
  }

print(signal)
```

```
{'name': 'Wnt', 'pathway': 'Canonical'}
```

Mutability

Dictionaries are mutable. This means that we can alter their contents. We can make any alterations to a dictionary as long as we use *immutable* values for the *keys*.

Suppose we have a dictionary stored in a variable called `protein`, holding some information about a specific protein:

```
protein = {
  'full name': 'Cystic fibrosis transmembrane conductance regulator',
  'alias': 'CFTR',
  'gene': 'CFTR',
  'type': 'Membrane Protein',
  'common mutations': ['Delta-F508', 'G542X', 'G551D', 'N1303K']
  }
```

We can add new *items* to our dictionary or alter the existing ones:

```
# Adding a new item:
protein['chromosome'] = 7

print(protein)

print(protein['chromosome'])
```

```
{'full name': 'Cystic fibrosis transmembrane conductance regulator', '
    alias': 'CFTR', 'gene': 'CFTR', 'type': 'Membrane Protein', 'common
    mutations': ['Delta-F508', 'G542X', 'G551D', 'N1303K'], 'chromosome'
    : 7}
```

```
7
```

We can also alter an existing *value* in a dictionary using its *key*. To do so, we simply access the *value* using its *key*, and treat it as a normal variable; i.e. the same way we do with members of a `list`:

```
print(protein['common mutations'])
```

```
['Delta-F508', 'G542X', 'G551D', 'N1303K']
```

```
protein['common mutations'].append('W1282X')
print(protein)
```

```
{'full name': 'Cystic fibrosis transmembrane conductance regulator', '
    alias': 'CFTR', 'gene': 'CFTR', 'type': 'Membrane Protein', 'common
    mutations': ['Delta-F508', 'G542X', 'G551D', 'N1303K', 'W1282X'], '
    chromosome': 7}
```

Do it Yourself

Implement the following dictionary:

```
signal = {'name': 'Wnt', 'pathway': 'Non-Canonical'}}
```

with respect to signal:

- Correct the *value* of pathway to “Canonical”;
- Add a new *item* to the dictionary to represent the *receptors* for the canonical pathway as “Frizzled” and “LRP”.

Display the altered dictionary as the final result.

```
signal = {'name': 'Wnt', 'pathway': 'Non-Canonical'}

signal['pathway'] = 'Canonical'
signal['receptors'] = ('Frizzled', 'LRP')

print(signal)
```

```
{'name': 'Wnt', 'pathway': 'Canonical', 'receptors': ('Frizzled', 'LRP'
    )}
```

Advanced Topic

Displaying an entire dictionary using the print() function can look a little messy because it is not properly structured. There is, however, an external library called `pprint` (Pretty-Print) that behaves

in very similar way to the default print() function, but structures dictionaries and other arrays in a more presentable way before displaying them. We do not discuss “Pretty-Print’ ’ in this course, but it is a part of Python’s default library and is therefore installed with Python automatically. To learn more it, have a read through the official documentations for the library and review the examples.

Because the *keys* are immutable, they cannot be altered. However, we can get around this limitation by introducing a new *key* and assigning the *values* of the old *key* to the new one. Once we do that, we can go ahead and *remove* the old *item*. The easiest way to remove an *item* from a dictionary is to use the syntax del:

```
# Creating a new key and assigning to it the
# values of the old key:
protein['human chromosome'] = protein['chromosome']

print(protein)
```

```
{'full name': 'Cystic fibrosis transmembrane conductance regulator', '
    alias': 'CFTR', 'gene': 'CFTR', 'type': 'Membrane Protein', 'common
    mutations': ['Delta-F508', 'G542X', 'G551D', 'N1303K', 'W1282X'], '
    chromosome': 7, 'human chromosome': 7}
```

```
# Now we remove the old item from the dictionary:
del protein['chromosome']

print(protein)
```

```
{'full name': 'Cystic fibrosis transmembrane conductance regulator', '
    alias': 'CFTR', 'gene': 'CFTR', 'type': 'Membrane Protein', 'common
    mutations': ['Delta-F508', 'G542X', 'G551D', 'N1303K', 'W1282X'], '
    human chromosome': 7}
```

We can simplify the above operation using the .pop() method, which removes the specified *key* from a dictionary and returns any *values* associated with it:

```
protein['common mutations in caucasians'] = protein.pop('common
    mutations')

print(protein)
```

```
{'full name': 'Cystic fibrosis transmembrane conductance regulator', '
    alias': 'CFTR', 'gene': 'CFTR', 'type': 'Membrane Protein', 'human
    chromosome': 7, 'common mutations in caucasians': ['Delta-F508', '
    G542X', 'G551D', 'N1303K', 'W1282X']}
```

Do it Yourself

Implement a dictionary as:

```
signal = {'name': 'Beta-Galactosidase', 'pdb': '4V40'}
```

with respect to signal:

- Change the *key* name from 'pdb' to 'pdb id' using the .pop() method.
- Write a code to find out whether the dictionary:
  - contains the new *key* (i.e. 'pdb id').
  - confirm that it no longer contains the old *key* (i.e. 'pdb')

If both conditions are met, display:

```
Contains the new key, but not the old one.
```

Otherwise:

```
Failed to alter the dictionary.
```

```
signal = {
            'name': 'Beta-Galactosidase',
            'pdb': '4V40'
    }

signal['pdb id'] = signal.pop('pdb')

if 'pdb id' in signal and 'pdb' not in signal:
    print('Contains the new key, but not the old one.')
else:
    print('Failed to alter the dictionary.')
```

```
Contains the new key, but not the old one.
```

Nested dictionaries

As explained earlier the section, dictionaries are amongst the most powerful built-in tools in Python. It is possible to construct nested dictionaries to organise data in a hierarchical fashion. This useful technique is outlined extensively in example.

It is very easy to implement nested dictionaries:

```
# Parent dictionary
pkc_family = {
```

```
    # Child dictionary A:
    'conventional': {
      'note': 'Require DAG, Ca2+, and phospholipid for activation.',
      'types': ['alpha', 'beta-1', 'beta-2', 'gamma']
    },
    # Child dictionary B:
    'atypical': {
        'note': (
            'Require neither Ca2+ nor DAG for'
            'activation (require phosphatidyl serine).'
        ),
        'types': ['iota', 'zeta']
    }
}
```

and we follow similar principles to access, alter, or remove the *values* stored in nested dictionaries:

```
print(pkc_family)
```

```
{'conventional': {'note': 'Require DAG, Ca2+, and phospholipid for
   activation.', 'types': ['alpha', 'beta-1', 'beta-2', 'gamma']}, '
   atypical': {'note': 'Require neither Ca2+ nor DAG foractivation (
   require phosphatidyl serine).', 'types': ['iota', 'zeta']}}
```

```
print(pkc_family['atypical'])
```

```
{'note': 'Require neither Ca2+ nor DAG foractivation (require
   phosphatidyl serine).', 'types': ['iota', 'zeta']}
```

```
print(pkc_family['conventional']['note'])
```

```
Require DAG, Ca2+, and phospholipid for activation.
```

```
print(pkc_family['conventional']['types'])
```

```
['alpha', 'beta-1', 'beta-2', 'gamma']
```

```
print(pkc_family['conventional']['types'][2])
```

```
beta-2
```

```
apkc_types = pkc_family['conventional']['types']
print(apkc_types[1])
```

```
beta-1
```

Do it Yourself

Implement the following table of genetic disorders as a nested dictionary:

	Full Name	Gene	Type
Cystic fibrosis	Cystic fibrosis trans-membrane conduc-tance regulator	CFTR	Membrane Protein
Xeroderma pigmentosum A	DNA repair protein comple-menting XP-A cells	XPA	Nucleotide excision repair
Haemophilia A	Haemophilia A	F8	Factor VIII Blood-clotting protein

Using the dictionary, display the *gene* for *Haemophilia A*.

```
genetic_diseases = {
    'Cystic fibrosis': {
        'name': 'Cystic fibrosis transmembrane conductance regulator',
        'gene': 'CFTR',
        'type': 'Membrane Protein'
    },
    'Xeroderma pigmentosum A': {
        'name': 'DNA repair protein complementing XP-A cells',
        'gene': 'XPA',
        'type': 'Nucleotide excision repair'
    },
    'Haemophilia A': {
        'name': 'Haemophilia A',
        'gene': 'F8',
```

```
        'type': 'Factor VIII Blood-clotting protein'
    }
}

print(genetic_diseases['Haemophilia A']['gene'])
```

```
F8
```

EXAMPLE: Nested dictionaries in practice

We would like to store and analyse the structure of several proteins involved in the *Lac operon*. To do so, we create a Python `dict` to help us organise our data.

We start off by creating an empty dictionary that will store our structures:

```
structures = dict()
```

We then move onto depositing our individual entries to structure by adding new *items* to it.

Each *item* has a *key* that represents the name of the protein we are depositing, and a *value* that is itself a dictionary consisting of information regarding the structure of that protein:

```
structures['Beta-Galactosidase'] = {
    'pdb id': '4V40',
    'deposit date': '1994-07-18',
    'organism': 'Escherichia coli',
    'method': 'x-ray',
    'resolution': 2.5,
    'authors': (
        'Jacobson, R.H.', 'Zhang, X.',
        'Dubose, R.F.', 'Matthews, B.W.'
    )
}
```

```
structures['Lactose Permease'] = {
    'pdb id': '1PV6',
    'deposit data': '2003-06-23',
    'organism': 'Escherichia coli',
    'method': 'x-ray',
    'resolution': 3.5,
    'authors': (
        'Abramson, J.', 'Smirnova, I.', 'Kasho, V.',
        'Verner, G.', 'Kaback, H.R.', 'Iwata, S.'
    )
}
```

Dictionaries don't have to be homogeneous. In other words, there can be different *items* in each entry.

For instance, the 'LacY' protein contains an additional *key* entitled 'note':

```
structures['LacY'] = {
    'pdb id': '2Y5Y',
    'deposit data': '2011-01-19',
    'organism': 'Escherichia coli',
    'method': 'x-ray',
    'resolution': 3.38,
    'note': 'in complex with an affinity inactivator',
    'authors': (
        'Chaptal, V.', 'Kwon, S.', 'Sawaya, M.R.',
        'Guan, L.', 'Kaback, H.R.', 'Abramson, J.'
    )
}
```

The variable structure which is an instance of type `dict`, is now a nested dictionary:

```
print(structures)
```

```
{'Beta-Galactosidase': {'pdb id': '4V40', 'deposit date': '1994-07-18',
    'organism': 'Escherichia coli', 'method': 'x-ray', 'resolution':
   2.5, 'authors': ('Jacobson, R.H.', 'Zhang, X.', 'Dubose, R.F.', '
   Matthews, B.W.')}, 'Lactose Permease': {'pdb id': '1PV6', 'deposit
   data': '2003-06-23', 'organism': 'Escherichia coli', 'method': 'x-
   ray', 'resolution': 3.5, 'authors': ('Abramson, J.', 'Smirnova, I.',
    'Kasho, V.', 'Verner, G.', 'Kaback, H.R.', 'Iwata, S.')}, 'LacY': {
   'pdb id': '2Y5Y', 'deposit data': '2011-01-19', 'organism': '
   Escherichia coli', 'method': 'x-ray', 'resolution': 3.38, 'note': '
   in complex with an affinity inactivator', 'authors': ('Chaptal, V.',
    'Kwon, S.', 'Sawaya, M.R.', 'Guan, L.', 'Kaback, H.R.', 'Abramson,
   J.')}}
```

We know that we can extract information from our nested `dict` just like we would with any other `dict`:

```
print(structures['Beta-Galactosidase'])
```

```
{'pdb id': '4V40', 'deposit date': '1994-07-18', 'organism': '
   Escherichia coli', 'method': 'x-ray', 'resolution': 2.5, 'authors':
   ('Jacobson, R.H.', 'Zhang, X.', 'Dubose, R.F.', 'Matthews, B.W.')}
```

```
print(structures['Beta-Galactosidase']['method'])
```

```
x-ray
```

```
print(structures['Beta-Galactosidase']['authors'])
```

```
('Jacobson, R.H.', 'Zhang, X.', 'Dubose, R.F.', 'Matthews, B.W.')
```

```
print(structures['Beta-Galactosidase']['authors'][0])
```

```
Jacobson, R.H.
```

Sometimes, especially when creating longer dictionaries, it might be easier to store individual entries in a variable beforehand and add them to the parent dictionary later on.

Note that our parent dictionary in this case is represented by the variable structure.

```
entry = {
    'Lac Repressor': {
        'pdb id': '1LBI',
        'deposit data': '1996-02-17',
        'organism': 'Escherichia coli',
        'method': 'x-ray',
        'resolution': 2.7,
        'authors': (
            'Lewis, M.', 'Chang, G.', 'Horton, N.C.',
            'Kercher, M.A.', 'Pace, H.C.', 'Lu, P.'
        )
    }
}
```

We can then use the .update() method to update our structures dictionary:

```
structures.update(entry)

print(structures['Lac Repressor'])
```

```
{'pdb id': '1LBI', 'deposit data': '1996-02-17', 'organism': '
    Escherichia coli', 'method': 'x-ray', 'resolution': 2.7, 'authors':
    ('Lewis, M.', 'Chang, G.', 'Horton, N.C.', 'Kercher, M.A.', 'Pace, H
    .C.', 'Lu, P.')}
```

We sometimes need to see what *keys* our dictionary contains. To obtain an array of *keys*, we use the method .keys() as follows:

```
print(structures.keys())
```

```
dict_keys(['Beta-Galactosidase', 'Lactose Permease', 'LacY', 'Lac
    Repressor'])
```

Likewise, we can also obtain an array of *values* in a dictionary using the .values() method:

```
print(structures['LacY'].values())
```

```
dict_values(['2Y5Y', '2011-01-19', 'Escherichia coli', 'x-ray', 3.38, '
    in complex with an affinity inactivator', ('Chaptal, V.', 'Kwon, S.'
    , 'Sawaya, M.R.', 'Guan, L.', 'Kaback, H.R.', 'Abramson, J.')])
```

We can then extract specific information to conduct an analysis. Note that the len() function in this context returns the number of *keys* in the parent dictionary only.

```
sum_resolutions = 0
res = 'resolution'

sum_resolutions += structures['Beta-Galactosidase'][res]
sum_resolutions += structures['Lactose Permease'][res]
sum_resolutions += structures['Lac Repressor'][res]
sum_resolutions += structures['LacY'][res]

total_entries = len(structures)

average_resolution = sum_resolutions / total_entries

print(average_resolution)
```

```
3.0199999999999996
```

Useful methods for dictionary

Now we use some snippets to demonstrate some of the useful *methods* associated with `dict` in Python.

Given a dictionary as:

```
lac_repressor = {
        'pdb id': '1LBI',
        'deposit data': '1996-02-17',
        'organism': 'Escherichia coli',
        'method': 'x-ray',
        'resolution': 2.7,
}
```

We can create an array of all *items* in the dictionary using the .items() method:

```
print(lac_repressor.items())
```

```
dict_items([('pdb id', '1LBI'), ('deposit data', '1996-02-17'), ('
    organism', 'Escherichia coli'), ('method', 'x-ray'), ('resolution',
    2.7)])
```

Similar to the enumerate() function (discussed in subsection DIY), the .items() method also returns an array of `tuple` members. Each `tuple` itself consists of 2 members, and is structured as ('key': 'value'). On that account, we can use its output in the context of a **`for`**–loop as follows:

```
for key, value in lac_repressor.items():
    print(key, value, sep=': ')
```

```
pdb id: 1LBI
deposit data: 1996-02-17
organism: Escherichia coli
method: x-ray
resolution: 2.7
```

Do it Yourself

Try .items() on a nested `dict` and see how it works.

```
nested_dict = {
    'L1-a': {
        'L2-Ka': 'L2_Va',
        'L2-Kb': 'L2_Vb',
    },
    'L1-b': {
        'L2-Kc': 'L2_Vc',
        'L2-Kd': 'L3_Vd'
    },
    'L3-c': 'L3_V'
}

print(nested_dict.items())
```

```
dict_items([('L1-a', {'L2-Ka': 'L2_Va', 'L2-Kb': 'L2_Vb'}), ('L1-b', {'
    L2-Kc': 'L2_Vc', 'L2-Kd': 'L3_Vd'}), ('L3-c', 'L3_V')])
```

We learned earlier that if we ask for a *key* that is not in the `dict`, a `KeyError` will be raised. If we anticipate this, we can handle it using the .get() method. The method takes in the *key* and searches the dictionary to find it. If found, the associating *value* is returned. Otherwise, the method returns `None` by default. We can also pass a second value to .get() to replace `None` in cases that the requested *key* does not exist:

```
print(lac_repressor['gene'])
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): KeyError: '
    gene'

Detailed traceback:
  File "<string>", line 1, in <module>
```

```
print(lac_repressor.get('gene'))
```

```
None
```

```
print(lac_repressor.get('gene', 'Not found...'))
```

```
Not found...
```

Do it Yourself

Implement the lac_repressor dictionary and try to extract the *values* associated with the following *keys*:

- organism
- authors
- subunits
- method

If a *key* does not exist in the dictionary, display No entry instead.

Display the results in the following format:

```
organism: XXX
authors: XXX
```

```
lac_repressor = {
    'pdb id': '1LBI',
    'deposit data': '1996-02-17',
    'organism': 'Escherichia coli',
    'method': 'x-ray',
    'resolution': 2.7,
}

requested_keys = ['organism', 'authors', 'subunits', 'method']

for key in requested_keys:
    lac_repressor.get(key, 'No entry')
```

```
'Escherichia coli'
'No entry'
'No entry'
'x-ray'
```

`for`-loop and dictionary

Dictionaries and **`for`**-loops create a powerful combination. We can leverage the accessibility of dictionary *values* through specific *keys* that we define ourselves in a loop to extract data iteratively and repeatedly.

One of the most useful tools that we can create using nothing more than a **`for`**-loop and a dictionary, in only a few lines of code, is a sequence converter.

Here, we are essentially iterating through a sequence of DNA nucleotides (sequence), extracting one character per loop cycle from our string (nucleotide). We then use that character as a *key* to retrieve its corresponding *value* from our a dictionary (dna2rna). Once we get the *value*, we add it to the variable that we initialised using an empty string outside the scope of our **`for`**-loop (rna_sequence) as discussed in subsection. At the end of the process, the variable rna_sequence will contain a converted version of our sequence.

```
1  sequence = '
       CCCATCTTAAGACTTCACAAGACTTGTGAAATCAGACCACTGCTCAATGCGGAACGCCCG'
2
3  dna2rna = {"A": "U", "T": "A", "C": "G", "G": "C"}
4
5  rna_sequence = str()  # Creating an empty string.
6
7  for nucleotide in sequence:
8      rna_sequence += dna2rna[nucleotide]
9
10 print('DNA:', sequence)
11 print('RNA:', rna_sequence)
```

```
1  DNA: CCCATCTTAAGACTTCACAAGACTTGTGAAATCAGACCACTGCTCAATGCGGAACGCCCG
2  RNA: GGGUAGAAUUCUGAAGUGUUCUGAACACUUUAGUCUGGUGACGAGUUACGCCUUGCGGGC
```

Do it Yourself

We know that in reverse transcription, RNA nucleotides are converted to their complementary DNA as shown:

Type	Direction	Nucleotides
RNA	5'…'	U A G C
cDNA	5'…'	A T C G

with that in mind:

1. Use the table to construct a dictionary for reverse transcription, and another dictionary for the conversion of cDNA to DNA.
2. Using the appropriate dictionary, convert the following mRNA (exon) sequence for human G protein-coupled receptor to its cDNA.

```
human_gpcr = (
    '
        AUGGAUGUGACUUCCCAAGCCCGGGGCGUGGGCCUGGAGAUGUACCCAGGCACCGCGCAGCCUGCGGCCCCCAACA
        '
    '
        CCCCGAGCUCAACCUGUCCCACCCGCUCCUGGGCACCGCCCUGGCCAAUGGGACAGGUGAGCUCUCGGAGCACCAG
        '
    '
        UGAUCGGCCUGUUCCUCUCGUGCCUCUACACCAUCUUCCUCUUCCCCAUCGGCUUUGUGGGCAACAUCCUGAUCCU
        '
    '
        AACAUCAGCUUCCGCGAGAAGAUGACCAUCCCCGACCUGUACUUCAUCAACCUGGCGGUGGCGGACCUCAUCCUGG
        '
    '
        CUCCCUCAUUGAGGUGUUCAACCUGCACGAGCGGUACUACGACAUCGCCGUCCUGUGCACCUUCAUGUCGCUCUUC
        '
    '
        UCAACAUGUACAGCAGCGUCUUCUUCCUCACCUGGAUGAGCUUCGACCGCUACAUCGCCCUGGCCAGGGCCAUGCG
        '
    '
        CUGUUCCGCACCAAGCACCACGCCCGGCUGAGCUGUGGCCUCAUCUGGAUGGCAUCCGUGUCAGCCACGCUGGUGC
        '
    '
        CGCCGUGCACCUGCAGCACACCGACGAGGCCUGCUUCUGUUUCGCGGAUGUCCGGGAGGUGCAGUGGCUCGAGGUC
        '
    '
        GCUUCAUCGUGCCCUUCGCCAUCAUCGGCCUGUGCUACUCCCUCAUUGUCCGGGUGCUGGUCAGGGCGCACCGGCA
        '
    '
        CUGCGGCCCCGGCGGCAGAAGGCGCUCCGCAUGAUCCUCGCGGUGGUGCUGGUCUUCUUCGUCUGCUGGCUGCCGG
        '
    '
        CUUCAUCAGCGUGCACCUCCUGCAGCGGACGCAGCCUGGGGCCGCUCCCUGCAAGCAGUCUUUCCGCCAUGCCCAC
        '
    '
        CGGGCCACAUUGUCAACCUCACCGCCUUCUCCAACAGCUGCCUAAACCCCCUCAUCUACAGCUUUCUCGGGGAGAC
        '
    '
        GACAAGCUGAGGCUGUACAUUGAGCAGAAAACAAAUUUGCCGGCCCUGAACCGCUUCUGUCACGCUGCCCUGAAGG
        '
    'UCCAGACAGCACCGAGCAGUCGGAUGUGAGGUUCAGCAGUGCCGUG'
)
```

```
mrna2cdna = {
```

```
    'U': 'A',
    'A': 'T',
    'G': 'C',
    'C': 'G'
}

cdna2dna = {
    'A': 'T',
    'T': 'A',
    'C': 'G',
    'G': 'C'
}
```

Q2

```
cdna = str()
for nucleotide in human_gpcr:
    cdna += mrna2cdna[nucleotide]

print(cdna)
```

```
TACCTACACTGAAGGGTTCGGGCCCCGCACCCGGACCTCTACATGGGTCCGTGGCGCGTCGGACGCCGGGGGTTGTGGTGGAG
```

Summary

In this section we talked about dictionaries, which are one the most powerful built-in types in Python. We learned:

- how to create dictionaries in Python,
- methods to alter or manipulate normal and nested dictionaries,
- two different techniques for changing an existing *key*,
- examples on how dictionaries help us organise our data and retrieve them when needed,

Finally, we also learned that we can create an *iterable* (discussed in section) from dictionary *keys* or *values* using the .key(), the .values(), or the .items() methods.

Exercises

End of chapter Exercises

We know that the process of protein translation starts by transcribing a gene from DNA to RNA *nucleotides*, followed by translating the RNA *codons* to protein.

Conventionally, we write a DNA sequence from the 5'-end to the 3'-end. The transcription process, however, starts from the 3'-end of a gene to the 5'-end (anti-sense strand), resulting in a sense mRNA sequence complementing the sense DNA strand. This is because RNA polymerase can only add nucleotides to the 3'-end of the growing mRNA chain, which eliminates the need for the Okazaki fragments as seen in DNA replication.

Example: The DNA sequence ATGTCTAAA is transcribed into AUGUCUAAA.

Given a conversion table:

DNA	A	T	C	G
cDNA	T	A	G	C
RNA	A	U	C	G

and this 5'- to 3'-end DNA sequence of 717 nucleotides for the Green Fluorescent Protein (GFP) mutant 3 extracted from Aequorea victoria:

```
dna_sequence = (
    'ATGTCTAAAGGTGAAGAATTATTCACTGGTGTTGTCCCAATTTTGGTTGAATTAGATGGTGATGTTAATGGT'
    'CACAAATTTTCTGTCTCCGGTGAAGGTGAAGGTGATGCTACTTACGGTAAATTGACCTTAAAATTTATTTGT'
    'ACTACTGGTAAATTGCCAGTTCCATGGCCAACCTTAGTCACTACTTTCGGTTATGGTGTTCAATGTTTTGCT'
    'AGATACCCAGATCATATGAAACAACATGACTTTTTCAAGTCTGCCATGCCAGAAGGTTATGTTCAAGAAAGA'
    'ACTATTTTTTTCAAAGATGACGGTAACTACAAGACCAGAGCTGAAGTCAAGTTTGAAGGTGATACCTTAGTT'
    'AATAGAATCGAATTAAAAGGTATTGATTTTAAAGAAGATGGTAACATTTTAGGTCACAAATTGGAATACAAC'
    'TATAACTCTCACAATGTTTACATCATGGCTGACAAACAAAAGAATGGTATCAAAGTTAACTTCAAAATTAGA
```

```
        '
    '
        CACAACATTGAAGATGGTTCTGTTCAATTAGCTGACCATTATCAACAAAATACTCCAATTGGTGATGGTCCA
        '
    '
        GTCTTGTTACCAGACAACCATTACTTATCCACTCAATCTGCCTTATCCAAAGATCCAAACGAAAAGAGAGAC
        '
    '
        CACATGGTCTTGTTAGAATTTGTTACTGCTGCTGGTATTACCCATGGTATGGATGAATTGTACAAATAA
        '
)
```

Use the DNA sequence and the conversion table to:

1. Write a Python script to *transcribe* this sequence to mRNA as it occurs in a biological organism. That is, determine the complimentary DNA first, and use that to work out the mRNA.
2. Use the following dictionary in a Python script to obtain the translation (protein sequence) of the Green Fluorescent Protein using the mRNA sequence you obtained.

```
codon2aa = {
    "UUU": "F", "UUC": "F", "UUA": "L", "UUG": "L", "CUU": "L",
    "CUC": "L", "CUA": "L", "CUG": "L", "AUU": "I", "AUC": "I",
    "AUA": "I", "GUU": "V", "GUC": "V", "GUA": "V", "GUG": "V",
    "UCU": "S", "UCC": "S", "UCA": "S", "UCG": "S", "AGU": "S",
    "AGC": "S", "CCU": "P", "CCC": "P", "CCA": "P", "CCG": "P",
    "ACU": "T", "ACC": "T", "ACA": "T", "ACG": "T", "GCU": "A",
    "GCC": "A", "GCA": "A", "GCG": "A", "UAU": "Y", "UAC": "Y",
    "CAU": "H", "CAC": "H", "CAA": "Q", "CAG": "Q", "AAU": "N",
    "AAC": "N", "AAA": "K", "AAG": "K", "GAU": "D", "GAC": "D",
    "GAA": "E", "GAG": "E", "UGU": "C", "UGC": "C", "UGG": "W",
    "CGU": "R", "CGC": "R", "CGA": "R", "CGG": "R", "AGA": "R",
    "AGG": "R", "GGU": "G", "GGC": "G", "GGA": "G", "GGG": "G",
    "AUG": "<Met>", "UAA": "<STOP>", "UAG": "<STOP>", "UGA": "<STOP>"
}
```

Q1

```
dna_sequence = (
    '
        ATGTCTAAAGGTGAAGAATTATTCACTGGTGTTGTCCCAATTTTGGTTGAATTAGATGGTGATGTTAATGGT
        '
    '
        CACAAATTTTCTGTCTCCGGTGAAGGTGAAGGTGATGCTACTTACGGTAAATTGACCTTAAAATTTATTTGT
        '
    '
        ACTACTGGTAAATTGCCAGTTCCATGGCCAACCTTAGTCACTACTTTCGGTTATGGTGTTCAATGTTTTGCT
        '
```

```
    '
        AGATACCCAGATCATATGAAACAACATGACTTTTTCAAGTCTGCCATGCCAGAAGGTTATGTTCAAGAAAGA
        '
    '
        ACTATTTTTTTCAAAGATGACGGTAACTACAAGACCAGAGCTGAAGTCAAGTTTGAAGGTGATACCTTAGTT
        '
    '
        AATAGAATCGAATTAAAAGGTATTGATTTTAAAGAAGATGGTAACATTTTAGGTCACAAATTGGAATACAAC
        '
    '
        TATAACTCTCACAATGTTTACATCATGGCTGACAAACAAAAGAATGGTATCAAAGTTAACTTCAAAATTAGA
        '
    '
        CACAACATTGAAGATGGTTCTGTTCAATTAGCTGACCATTATCAACAAAATACTCCAATTGGTGATGGTCCA
        '
    '
        GTCTTGTTACCAGACAACCATTACTTATCCACTCAATCTGCCTTATCCAAAGATCCAAACGAAAAGAGAGAC
        '
    '
        CACATGGTCTTGTTAGAATTTGTTACTGCTGCTGGTATTACCCATGGTATGGATGAATTGTACAAATAA
        '
)


codon2aa = {
    "UUU": "F", "UUC": "F", "UUA": "L", "UUG": "L", "CUU": "L",
    "CUC": "L", "CUA": "L", "CUG": "L", "AUU": "I", "AUC": "I",
    "AUA": "I", "GUU": "V", "GUC": "V", "GUA": "V", "GUG": "V",
    "UCU": "S", "UCC": "S", "UCA": "S", "UCG": "S", "AGU": "S",
    "AGC": "S", "CCU": "P", "CCC": "P", "CCA": "P", "CCG": "P",
    "ACU": "T", "ACC": "T", "ACA": "T", "ACG": "T", "GCU": "A",
    "GCC": "A", "GCA": "A", "GCG": "A", "UAU": "Y", "UAC": "Y",
    "CAU": "H", "CAC": "H", "CAA": "Q", "CAG": "Q", "AAU": "N",
    "AAC": "N", "AAA": "K", "AAG": "K", "GAU": "D", "GAC": "D",
    "GAA": "E", "GAG": "E", "UGU": "C", "UGC": "C", "UGG": "W",
    "CGU": "R", "CGC": "R", "CGA": "R", "CGG": "R", "AGA": "R",
    "AGG": "R", "GGU": "G", "GGC": "G", "GGA": "G", "GGG": "G",
    "AUG": "<Met>", "UAA": "<STOP>", "UAG": "<STOP>", "UGA": "<STOP>"
}

dna2cdna = {
    'A': 'T',
    'C': 'G',
    'G': 'C',
    'T': 'A'
}


dna2mrna = {
    'A': 'U',
    'T': 'A',
```

```
    'G': 'C',
    'C': 'G'
}


# Transcription
# ---------------------------------------------------------------------
m_rna = str()

for nucleotide in dna_sequence:
    # DNA to cDNA
    c_dna = dna2cdna[nucleotide]

    # cDNA to mRNA
    m_rna += dna2mrna[c_dna]


print('mRNA:', m_rna)


# Translation:
# ---------------------------------------------------------------------
```

```
mRNA:
    AUGUCUAAAGGUGAAGAAUUAUUCACUGGUGUUGUCCCAAUUUUGGUUGAAUUAGAUGGUGAUGUUAAUGGUCACAAAUU
```

```
mrna_len = len(m_rna)
codon_len = 3

protein = str()

for index in range(0, mrna_len, codon_len):
    codon = m_rna[index: index + codon_len]
    protein += codon2aa[codon]

print('Protein:', protein)

# ---------------------------------------------------------------------
# INTERMEDIATE-LEVEL TWIST (Alternative answer):
# One can also combine the two processes.
#
# Advantages:
#   - One for-loop.
#   - No use of `range()`.
#   - Almost twice as fast (half as many iterations).
# ---------------------------------------------------------------------
```

```
Protein: <Met>
    SKGEELFTGVVPILVELDGDVNGHKFSVSGEGEGDATYGKLTLKFICTTGKLPVPWPTLVTTFGYGVQCFARYPDH
```

```
    <Met>KQHDFFKSA<Met>
    PEGYVQERTIFFKDDGNYKTRAEVKFEGDTLVNRIELKGIDFKEDGNILGHKLEYNYNSHNVYI<Met
    >ADKQKNGIKVNFKIRHNIEDGSVQLADHYQQNTPIGDGPVLLPDNHYLSTQSALSKDPNEKRDH<
    Met>VLLEFVTAAGITHG<Met>DELYK<STOP>
```

```
m_rna = str()
protein = str()
codon = str()

for nucleotide in dna_sequence:
    # DNA to cDNA
    c_dna = dna2cdna[nucleotide]

    # Transcription:
    transcribed_nucleotide = dna2mrna[c_dna]
    m_rna += transcribed_nucleotide

    # Translation process:
    # Retaining the residue to construct triplets.
    codon += transcribed_nucleotide

    # Check if this is a triplet (a codon):
    if len(codon) == 3:
        # Convert to amino acid and store:
        protein += codon2aa[codon]

        # Reset the codon to an empty string:
        codon = str()
print('mRNA:', m_rna)
```

```
mRNA:
    AUGUCUAAAGGUGAAGAAUUAUUCACUGGUGUUGUCCCAAUUUUGGUUGAAUUAGAUGGUGAUGUUAAUGGUCACAAAUU
```

```
print('Protein:', protein)
```

```
Protein: <Met>
    SKGEELFTGVVPILVELDGDVNGHKFSVSGEGEGDATYGKLTLKFICTTGKLPVPWPTLVTTFGYGVQCFARYPDH
    <Met>KQHDFFKSA<Met>
    PEGYVQERTIFFKDDGNYKTRAEVKFEGDTLVNRIELKGIDFKEDGNILGHKLEYNYNSHNVYI<Met
    >ADKQKNGIKVNFKIRHNIEDGSVQLADHYQQNTPIGDGPVLLPDNHYLSTQSALSKDPNEKRDH<
    Met>VLLEFVTAAGITHG<Met>DELYK<STOP>
```

Keypoints

- Dictionaries associate a set of *values* with a number of *keys*.
- *keys* are used to access the values of a dictionary.
- Dictionaries are mutable.

- Nested dictionaries are constructed to organise data in a hierarchical fashion.
- Some of the useful methods to work with dictionaries are: .items(), .get()