# Python 8

## Data Structures

Sometimes a *simple* list or dictionary just doesn't do what you want. Sometimes you need to organize data in a more *complex* way. You can nest any data type inside any other type. This lets you build multidimensional data tables easily.

### List of lists

List of lists, often called a matrix are important for organizing and accessing data

Here's a way to make a 3 x 3 table of values.

```
>>> M = [[1,2,3], [4,5,6],[7,8,9]]
>>> M[1] # second row (starts with index 0)
[4,5,6]
>>>M[1][2] # second row, third element
6
```

Here's a way to store sequence alignment data:

Four sequences aligned:

```
AT-TG
AATAG
T-TTG
AA-TA
```

The alignment in a list of lists.

```
aln = [['A', 'T', '-', 'T', 'G'],
['A', 'A', 'T', 'A', 'G'],
['T', '-', 'T', 'T', 'G'],
['A', 'A', '-', 'T', 'A']]
```

Get an the full length of one sequence:

```
>>> seq = aln[2]
>>> seq
['T', '-', 'T', 'T', 'G']
```

> Use the outer most index to access each sequence

Retrieve the nucleotide at a particular position in a sequence.

```
>>> nt = aln[2][3]
>>> nt
'T'
```

> Use the outer most index to access the sequence of interest and the inner most index to access the position

Get every nucleotide in a single column:

```
>>> col = [seq[3] for seq in aln]
>>> col
['T', 'A', 'T', 'T']
```

> Retrieve each sequence from the aln list then the 3rd column for each sequence.

## Lists of dictionaries

You can nest dictionaries in lists as well:

```
>>> records = [
... {'name' : 'actgctagt', 'accession' : 'ABC123', 'genetic_code' : 1},
... {'name' : 'ttaggttta', 'accession' : 'XYZ456', 'genetic_code' : 1},
... {'name' : 'cgcgatcgt', 'accession' : 'HIJ789', 'genetic_code' : 5}
... ]
>>> records[0]['name']
'actgctagt'
>>> records[0]['accession']
'ABC123'
>>> records[0]['genetic_code']
1
```

> Here you can retrieve the accession of one record at a time by using a combination of the outer index and the key 'accession'

## Dictionaries of lists

And, if you haven't guessed, you can nest lists in dictionaries

Here is a dictionary of kmers. The key is the kmer and its values is a list of postions

```
>>> kmers = {'ggaa': [4, 10], 'aatt': [0, 6, 12], 'gaat': [5, 11], 'tgga':
... [3, 9], 'attg': [1, 7, 13], 'ttgg': [2, 8]}
>>> kmers
{'tgga': [3, 9], 'ttgg': [2, 8], 'aatt': [0, 6, 12], 'attg': [1, 7, 13], 'ggaa':
[4, 10], 'gaat': [5, 11]}
>>>
>>> kmers['ggaa']
[4, 10]
>>> len(kmers['ggaa'])
2
```

> Here we can get a list of the positions of a kmer by using the kmer as the key. We can also do things to the returned list, like determining its length. The length will be the total count of this kmers.

You can also use the `get()` method to retrieve records.

```
>>> kmers['ggaa']
[4, 10]
>>> kmers.get('ggaa')
[4, 10]
```

> These two statements returns the same results, but if the key does not exist you will get nothing and not an error.

## Dictionaries of dictionaries

Dictionaries of dictionaries is my favorite!! You can do so many useful things with this data structure. Here we are storing a gene name and some different types of information about that gene, such as its, sequence, length, description, nucleotide composition and length.

```
>>> genes = {
... 'gene1' : {
...     'seq' : "TATGCC",
...    'desc' : 'something',
...     'len' : 6,
... 'nt_comp' : {
...             'A' : 1,
...             'T' : 2,
...             'G' : 1,
...             'C' : 2,
...            }
...   },
...
... 'gene2' : {
...     'seq' : "CAAATG",
...    'desc' : 'something',
...     'len' : 6,
... 'nt_comp' : {
...           'A' : 3,
...           'T' : 1,
...           'G' : 1,
...           'C' : 1,
...           }
...       }
... }
>>> genes
{'gene1': {'nt_comp': {'C': 2, 'G': 1, 'A': 1, 'T': 2}, 'desc': 'something',
'len': 6, 'seq': 'TATGCC'}, 'gene2': {'nt_comp': {'C': 1, 'G': 1, 'A': 3, 'T': 1},
'desc': 'something', 'len': 6, 'seq': 'CAAATG'}}
>>> genes['gene2']['nt_comp']
{'C': 1, 'G': 1, 'A': 3, 'T': 1}
```

> Here we store a gene name as the outermost key, with a second level of keys for qualities of the gene, like sequence, length, nucleotide composition. We can retrieve a quality by using the gene name and quality in conjunction.

There are also specific data table and frame handling libraries like Pandas. Here is a intro to data structures in Panda.

## Exceptions

There are a few different types of errors when coding. Syntax errors, logic errors, and exceptions. You have probably encountered all three. Sytax and logic errors are issues you need to deal with while coding. An exception is a special type of error that can be informative and used to write code to respond to this type of error. This is especially relavent when dealing with user input. What if

they don't give you any, or it is the wrong kind of input. We want our code to be able to detect these types of errors and respond accordingly.

```
#!/usr/bin/env python3

import sys
file = sys.argv[1]

print("User provided file:" , file)
```

> This code takes user provided input and prints it

Run it.

```
$ python scripts/exceptions.py test.txt
User provided file: test.txt
```

What happens if the user does not provide any input and we try to print it?

```
$ python scripts/exceptions.py
Traceback (most recent call last):
  File "scripts/exceptions.py", line 4, in <module>
    file = sys.argv[1]
IndexError: list index out of range
```

> We get an **IndexError** exception

We have already seen quite a few exceptions, here are some:

- ValueError: math domain error
- AttributeError: 'list' object has no attribute 'rstrip'
- SyntaxError: EOL while scanning string literal
- NameError: name 'GGTCTAC' is not defined
- SyntaxError: Missing parentheses in call to 'print'
- AttributeError: 'int' object has no attribute 'lower'
- IndexError: list assignment index out of range
- NameError: name 'HDAC' is not defined

We can use the exception to our advantage to help out our users. We can use a try/except condition to look for exceptions and to do something if we do not have an exception and do something different if we do have an exception.

```
#!/usr/bin/env python3
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file:" , file)
except:
  print("Please provide a file name")
```

> We need to "try" to get a user provided argument. If we are successful then we can print it out. If we try and fail, we execute the code in the except portion of our try/except and print that we need a file name.

Let's run it WITH iser input

```
$ python3 scripts/exceptions_try.py test.txt
User provided file: test.txt
```

> It runs as expected

Let's run it WITHOUT user input

```
$ python scripts/exceptions_try.py
Please provide a file name
```

> Yeah, the user is informed that they need to provide a file name to the script

What if the user provides input but it is not a valid file or the path is incorrect? Or if you want to check to see if the user provided input as well as if it can open the input.

We can add multiple exception tests. Each except statement can specify what kind of exception it is waiting to recieve. If that kind of exception occures, that block of code will be executed.

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  FASTA = open(file, "r")
  for line in FASTA:
    line = line.rstrip()
    print(line)
except IndexError:
  print("Please provide a file name")
except IOError:
  print("Can't find file:" , file)
```

> Here we test for an IndexError and a IOError. The IndexError occurs when we try to access a list element that does not exists. The IOError happens when we try to access a file that does not exist.

Let's run it with no input.

```
$ python scripts/exceptions_try_files.py test.txt
User provided file name: test.txt
Can't find file: test.txt
```

> This informs the user that they did provide input but that the file listed can not be found.

Let's run it with no input

```
$ python scripts/exceptions_try_files.py
Please provide a file name
```

> This informs the user that they need to provide a file.

## try/except/else/finally

Lets summarize what we have covered and add on `else` and `finally`.

```
try:
  # try block is executed until an exception is raised
except _ExceptionType_:
  # if there is an exception of "ExceptionType" this block will be executed
  # there can be more than one except block, just like an elif
except:
  # if there are any exceptions that are not of _ExceptionType_ this except block will be executed
else:
  # the else block is executed after the try block has been completed, which means there were no exceptions raised
finally:
  # the finally block is executed if exceptions are or are not raised (no matter what happens)
```

## Getting more information about an exception

Some exceptions can be thrown for multiple reasons, for example, ErrorIO will occur if the file does not exist as well as if you don't have permissions to read it. We can get more information by viewing the contents of our Exception Object. Yes, an exception is an object too! The system errors get stored in the exception object. To access the object use `as` and supply a variable name, like 'ex'

```
file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  FASTA = open(file, "r")
  for line in FASTA:
    line = line.rstrip()
    print(line)
except IndexError:
  print("Please provide a file name")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we added `except IOError as ex` and now we can get the 'strerror' message from ex.

Run it.

```
$ python scripts/exceptions_try_files_as.py  test.txt
User provided file name: test.txt
Can't find file: test.txt :  No such file or directory
```

> Now we know that this file name or path is not valid

## Raising an Exception

We can call or raise exceptions too!! This is accomplished by using a `raise` statement.

1. First, create a new Exception Object, i.e., `ValueError()`
2. Use the Exception Object in a Raise statment `raise ValueError('your message')`

Let's raise an exception if the file name does not end in 'fa'

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise ValueError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
except IndexError:
  print("Please provide a file name")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we raise a known exception, 'ValueError', if the file does not end with (uses `endswith()` method).

Let's run it.

```
$ python scripts/exceptions_try_files_raise.py test.txt
User provided file name: test.txt
Traceback (most recent call last):
  File "scripts/exceptions_try_files_raise.py", line 10, in <module>
    raise ValueError("Not a FASTA file")
ValueError: Not a FASTA file
```

> Our exception get's raised, now lets do something with it.

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise ValueError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
except IndexError:
  print("Please provide a file name")
except ValueError:
  print("File needs to be a FASTA file and end with .fa")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we created an except to catch any ValueError

Let's Run it.

```
$ python scripts/exceptions_try_files_raise_value.py test.txt
User provided file name: test.txt
File needs to be a FASTA file and end with .fa
```

> We get a great error message now.

But what if there is another ValueError, how can we tell if it is do to the FASTA file extension or not?

## Creating Custom Exceptions

We can create our own custom exception. We will need to create a new class of exception. Below is the sytax to do this.

```
import sys

class NotFASTAError(Exception):
  pass


file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise NotFASTAError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
except IndexError:
  print("Please provide a file name")
except NotFASTAError:
  print("File needs to be a FASTA file and end with .fa")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we created a new class of exception called 'NotFASTAError'. Then we raised this new exception.

Let's Run it.

```
$ python scripts/exceptions_try_files_raise_try.py test.txt
User provided file name: test.txt
File needs to be a FASTA file and end with .fa
```

> Our new class of exception, NotFASTAError, works just like the built in exceptions.