Week 4

Chapter 1

Outline

- Module Creation, Execution and Linting
- Module Execution
- Python Coding Style
- Lists
- Common Sequence Operations

Chapter 2

Module Creation, Execution and Linting

See the notes from Module Execution and Style Guidelines, the last two sections in Week3.

Install the Python package `pylint`:

```
% pip install pylint
```

We will follow the style guidelines to develop code for an assigned exercise, first in a notebook and then in a module. We will also see the benefits of linting the code to ensure that style guidelines are being adequately followed.

2.1 Exercise

Secure passwords are a first line of defense while protecting information stored on a device or stored online. Suppose we wish to device a program that validates a password through an interactive dialog. For our purposes, a valid password follows these rules:

- it is at least 10 characters long
- it contains at least one uppercase, one lowercase and one special character from among `@`, `#`, `$`, `%`, `^`, `&`, and `*`.
- it contains at least two occurrences of digits that are separated by other non-digit characters in the password.

Write a function `is_valid` that takes a single string-valued (formal) parameter `password` and returns `True` if the string conforms to a legal password as per our rules, or returns `False` otherwise.

Chapter 3

Sequence Datatypes

The **`str`** datatype is a **sequence datatype**, but it is immutable. Another immutable sequence type is **tuple**, which is a collection of fixed length or **arity**.

Cannot use assignment to modify elements in an immutable sequence.

The **`list`** datatype is also a sequence datatype. Lists are **mutable** collections: they can change over time.

> Caution
>
> Lists are **not** sets; there is a separate collection type called **set** in Python! Thus, an element can occur more than once in a list, but only occurs at most once in a set.

Chapter 4

List Creation

- `list()`: constructs an empty list and returns a reference to it
- The literal `[]`: this is *not* the preferred way to create a new list.

Many commonly used builtin library functions return lists:

- `<string>.split(<separator>)` will return a list of strings obtained by splitting `<string>` using the substring `<separator>` (the default is to split by whitespace)
- `<file_handle>.readlines()` returns a list of lines in a file being read (from the current read position in the file)

Chapter 5

List Operations

List elements are numbered from 0 to the length of the list - 1. These numbers are the **indices** (or *positions*).

> An **out-of-range** error called **IndexError** occurs if we try to use an index that does not exist for a list!

- `len(<list>)`: returns the length of list
- `<list>[<index>]`: returns the element at the given index in list
- `<list>[<slice>]`: returns a fresh list corresponding to the sliced portion
- `<list>.append(<element>)`: inserts `<element>` at the end of list
- `<list>.insert(<index>, <element>)`: a more general version of `append`; inserts just before the position given by `<index>`
- `<element> in <list>`: boolean expression for membership in list
- `<list>.index(<element>)`: returns the smallest index where element is found; else raises a **ValueError** exception
- `<list>.pop([<index>])`: removes the element at the given index (or, by default, the last element if no position is not specified); subsequent elements "move up" in the list.
- `list>.remove(<element>)`: removes the first occurrence (if any) of the element from list; raises **ValueError** if not found.

ℹ Note

Python has a bunch of builtin **exceptions** - these are errors that are **`raise`**d in code when something goes wrong. These exceptions have descriptive names that generally provide a clear idea as to what may have gone wrong: e.g., **AssertionError**, **ValueError**, , **IndexError**, **TypeError** etc.

Chapter 6

Example

- Find the cumulative sum of numbers in a list
- Read a CSV file
- Count the number of times that Tom Sawyer (‘Tom’) is referenced in various chapters of “Huckleberry Finn”

Chapter 7

List Aliasing

An assignment sequence

```
<var_1> = <list>
<var_2> = <var_1>
```

does not make a fresh copy of the list: it simply creates an *alias* or another named reference, `<var_2>` to the list. Any changes made via one of the references **affects** the value of the other!

```
list_1 = [1,2,3,4]
list_2 = list_1
list_3 = list_1[1:3]
list_1.append(5)
list_1[2] = 10
```

Chapter 8

Deep Copying

As structured datatype values become more complex (i.e. nested), it is even more important to think through the consequences of aliasing when these values are **copied**.

To make a “shallow” copy of the list, you can either do the following:

```
<var_2> = <var_1>[:]
```

or, import the `copy` builtin module and:

```
import copy
<var_2> = copy.copy(<var_1>)
```

For a “deep” copy:

```
import copy
<var_2> = copy.deepcopy(<var_1>)
```

Chapter 9

Sequence Operations

Python sequence types have several operations in common with other sequence types: the list of sequence operations includes slicing, concatenation, indexing and so on.

Chapter 10

Example

Read a CSV file of grades: it contains columns starting with the student name, and the grades on assignments. Compute the mean values of these scores.