Python Workshop

3. Data Structures

Data Structures

- Some commonly used data structures:
- **List** `[1, 2, 3]`
  - List of values, in a specific order.
- **Tuple** `(1, 2, 3)`
  - Immutable list of values.
- **Set** `{1, 2, 3}`
  - Unordered collection of unique values.
- **Dictionary** `{'a': 1, 'b': 2, 'c': 3}`
  - Unordered key-value pairs.

Defining a list

- A list of values can be defined using `[]`.

```
myList = [1, 2, 3]
```

- Values can be of mixed types, but it is **not recommended**.

```
myPoorList = [1, 2.3, 'abc']
```

Using a list

- You can output a list directly.

```
print('myList:', myList)
```

- **Size** of list can be retrived using `len()` function.

```
print('Length of list is', len(myList))
```

Accessing list item

- Getting **items** from list can be done using `[]` operator.

```
print('First item in the list is', myList[0])
print('Second item in the list is', myList[1])
print('Third item in the list is', myList[2])
```

- Note that **index** starts with `0`.

Append

- We can use `append()` to **add** one item to a list:

```
myList = []
myList.append(10)
myList.append(20)
myList.append(30)
print(myList)
```

Output:

```
[10, 20, 30]
```

- Note that `append()` modify the list directly.

Concatenate

- We can use the `+` operator to **concatenate** two lists:

```
a = [1, 2]
b = [3, 4]
print(a + b)
```

Output:

```
[1, 2, 3, 4]
```

- Note that both `a` and `b` must be lists.
- We will cover more operations in the next section.

For-loop revisited

- For-loop can be used to **iterate** through a list

```
sum = 0
myList = [1, 2, 3]
for i in myList:
    sum += i
print('sum =', sum)
```

Output:

```
sum = 6
```

Self-learning topics (~40min)

- List comprehensions
- Tuple
- Advanced topics:
    - Dictionary
    - Set
- You are encouraged to copy-and-paste the code to Spyder and test them.

List comprehensions

List comprehensions

- Suppose you are asked to **create** a list of values from 1 to 10, the easiest way is to append items to an empty list:

```
myList = []
for i in range(1, 11):
    myList.append(i)
print(myList)
```

- What are the other options?

Using range()

- We can convert `range()` to a list

```
myList = list(range(1, 11))
print(myList)
```

- Note that `range()` is not a list, it is an object that can be iterated.
- What if we want a list of square numbers instead?

Using for-loop

- Back to our old friend:

```
myList = []
for i in range(1, 11):
    myList.append(i * i)
print(myList)
```

The generator

- Alternatively, we can use the **generator** expression to generate a list

```
myList = [ i * i for i in range(1, 11) ]
print(myList)
```

- `for i in range( 1, 11 )` is equivalent to a for-loop.
- `i * i` is equivalent to the epxression appended to the list in the for-loop.

Generating list from list

- We can also generate a list from another list

```
list1 = [2, 3, 5, 7, 11]
list2 = [ x * 2 for x in list1 ]
print(list2)
```

Output:

```
[4, 6, 10, 14, 22]
```

Filtering a list

- A condition can be added in the generator to filter the result.

```
myList = [ x * x for x in range(1, 11) if x % 4 == 0 ]
print(myList)
```

Output:

```
[16, 64]
```

For-loop example

```
myList = [ x * x for x in range(1, 11) if x % 4 == 0 ]
```

- Same could be done using a for-loop and an if-statement:

```
myList = []
for x in range(1, 11):
    if x % 4 == 0:
      myList.append(x * x)
```

Exercises (1)

Exercise 1 (1)

- Write a program that reads two integers, first integer represents the **number of days** in a month, the second integer represents the **day of week** of the first day. Then generate a list with all Saturdays and Sundays and print it.
- Day of week is `0` for Sunday, `1` for Monday, etc.
- For example if the input is `30` and `1`, the output should be `[6, 7, 13, 14, 20, 21, 27, 28]`.

Exercise 1 (2)

- Sample input/output as follow:

Input	Output
28 4	[3, 4, 10, 11, 17, 18, 24, 25]
31 2	[5, 6, 12, 13, 19, 20, 26, 27]

Exercise 1 (3)

- More sample input/output:

Input	Output
30 6	[1, 2, 8, 9, 15, 16, 22, 23, 29, 30]
31 0	[1, 7, 8, 14, 15, 21, 22, 28, 29]

Exercise 2 (1)

- Write a program that repeatedly reads integers until a zero is received.
- In the process, create a **list** that keep all integers read.
- When the program receives a zero, print all items the list in reverse order (in any way you want).

Exercise 2 sample input/output

Input	Output
1 2 3 0	3, 2, 1

Input	Output
3 2 3 1 0	1, 3, 2, 3

Input	Output
1 1 1 1 0	1, 1, 1, 1

Tuple

Tuple

- A **tuple** is very similar to a list, except that `( )` is used instead of `[ ]` when defining it.

```
myTuple = (1, 2, 3, 4)
sum = 0
for i in myTuple:
    sum += i
print('sum =', sum)
```

Immutable

- Tuple is **immutable**, so the following will throw an exception.

```
myTuple = (1, 2, 3, 4)
myTuple[0] = 1
```

Automatic tuple packing

- A tuple without the bracket is an **expression list**.
- We can assign an expression list to a variable, the values will be automatically **packed** into a tuple.

```
a = 10
b = a, a + 1, a * 3
print(b)
```

Output:

```
(10, 11, 30)
```

Automatic tuple unpacking

- When we assign a tuple to a list of variables, the tuple will automatically be **unpacked**.

```
a = (37, 100)
b, c = a
print(b, c)
```

Output:

```
37 100
```

Packing and unpacking (1)

- Packing and unpacking can be done in the same statement. Therefore the following is possible:

```
a = 37
b = 100
b, a = a + b, b - a
print(a, b)
```

Output:

```
63 137
```

Packing and unpacking (2)

```
b, a = a + b, b - a
```

- The statement will be executed in two steps.
    1. Expression list `a + b, b - a` is temporarily packed to a tuple.
    2. The tuple is unpakced to variables `b` and `a` respsectively.

Empty tuple and singleton

- To create an empty tuple, simply use `()`:

```
a = ()
```

- To create a tuple with a single element, a comma is needed:

```
a = (1,)
```

- Without the comma, the bracket will be treated as a simple bracket as in an arithmetic expression.

Tuple generator

- The **generator** expression can also be used to generate a tuple:

```
myTuple = tuple( i * i for i in range(1, 11) )
print(myTuple)
```

- `tuple()` can also be used to convert a list to a tuple.

Exercises (2)

Exercise 3 (1)

- Write a program that reads two integers and calculate the **GCD** of the two integers using the following algorithm:
    1. Read integer as `a`.
    2. Read integer as `b`.
    3. While `b` is not zero; set `b` as `a % b`, and `a` as `b` at the same time.
    4. print `a` as the GCD.
- Make use of **tuple packing and unpacking** to complete the task.

Exercise 3 (2)

- Sample input/output as follow:

Input	Output	Input	Output
11 19	1	15 24	3
36 24	12	24 15	3

Advanced topics

- Remember: you may skip this section and come back later.

Dictionary

Dictionary

- A dictionary is a list with **key-value** pairs.
- Values are accessed by the key instead of an index.

```
d = {'a': 1, 'b': 2, 'c': 3}
print(d['a'])
print(d['b'])
print(d['c'])
```

`in` operator

- `in` is an operator that checks if an item is in a list/tuple/set.

```
fruits = ["apple", "orange", "banana"]
if "apple" in fruits:
    print('apple is in fruits!')
if "tomato" in fruits:
    print('tomato is in fruits!')
```

`in` operator of dictionary

- For dictionary, the `in` operator checks the key instead.

```python
fruits = {'apple': 100, 'orange': 200, 'banana': 300}
if "apple" in fruits:
    print('apple is in fruits!')
if "tomato" in fruits:
    print('tomato is in fruits!')
```

for-loop of dictionary

- When **for-loop** is used on a dictionary, the key is used instead of values.

```
fruits = {'apple': 100, 'orange': 200, 'banana': 300}
for f in fruits:
    print(f, fruits[f])
```

Output:

```
apple 100
orange 200
banana 300
```

Views of a dictionary

- We can get the **key view** or **value view** of a dictionary using `keys()` and `values()`:

```
fruits = {'apple': 100, 'orange': 200, 'banana': 300}
print(fruits.keys())
print(fruits.values())
```

Output:

```
dict_keys(['apple', 'orange', 'banana'])
dict_values([100, 200, 300])
```

- These can then be iterated in loops or generators.

Dictionary generator

- The **generator** expression can also be used to generate a dictionary:

```
myDict = { i: i * i for i in range(1, 8) }
print(myDict)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49}
```

Exercises (3)

Exercise 4 (1)

- Modify the program in Exercise 2 so that a **dictionary** is used to keep counts of odd and even integers received.
- When the program receives a zero, print the counts in any way you want.

Exercise 4 sample input/output

Input	Output
1	
2	odd: 2
3	even: 1
0	

Input	Output
3	
2	odd: 3
3	even: 1
1	
0	

Input	Output
1	
1	odd: 4
1	even: 0
1	
0	

Set

Set

- **Set** is defined using `{}`

```python
s = {1, 2, 3}
print(s)
```

- In a set, we only care if an item is in the set or not, the order is not guaranteed.

Set operations

- Basic **set operations** including union `|`, intersection `&`, difference `-`, and symmetric difference `^` are supported.

```
p = {1, 2, 3, 4}
q = {3, 4, 5, 6}
print(p | q, p & q)
print(p - q, p ^ q)
```

Output (order may be different):

```
{1, 2, 3, 4, 5, 6} {3, 4}
{1, 2} {1, 2, 5, 6}
```

Empty set

- We have to use the `set()` function to create an empty set.

```
s = set()
print(s)
```

- If you use `{}`, a dictionary is created instead.

Set generator

- The **generator** expression can also be used to generate a set:

```python
mySet = set(i * i for i in range(1, 11))
print(mySet)
```

Output:

```
{64, 1, 4, 36, 100, 9, 16, 49, 81, 25}
```